Реальная криптография

Дэвид Вонг



MANNING

Real-World Cryptography

DAVID WONG



MANNING
SHELTER ISLAND

Реальная криптография

Дэвид Вонг

ПИТЕР®

Санкт-Петербург · Москва · Минск

2024

Дэвид Вонг

Реальная криптография

Серия «Библиотека программиста»

Перевела с английского Д. Романовская

ББК 32.973.23-018-07
УДК 004.056.5

Вонг Дэвид

В73 Реальная криптография. — СПб.: Питер, 2024. — 432 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2091-8

Криптография — это фундамент информационной безопасности. Чтобы быть на шаг впереди хакеров, атакующих ваши системы, необходимо разбираться в инструментах, структурах и протоколах, защищающих ваши сети и приложения. В книге просто и наглядно описываются аутентификация, шифрование, подписи, хранение секретов и другие криптографические понятия.

Изучите приемы, которые используют разработчики, системные администраторы и специалисты по безопасности в своей повседневной работе. Вы не встретите ни сложных математических выкладок, ни умопомрачительного жаргона: современные криптографические методы рассматриваются с помощью понятных графиков и практических примеров. Познакомьтесь с хеш-функциями и подписями, криптографическими протоколами и безопасным обменом сообщениями, а также с передовыми достижениями, такими как постквантовая криптография и криптовалюты. Читать «Реальную криптографию» одно удовольствие и, возможно, она спасет, когда в следующий раз злоумышленники покусятся на ваши данные.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1617296710 англ.
ISBN 978-5-4461-2091-8

Authorized translation of the English edition © 2021 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.
© Перевод на русский язык ООО «Прогресс книга», 2024
© Издание на русском языке, оформление ООО «Прогресс книга», 2024
© Серия «Библиотека программиста», 2024

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.05.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 700 экз. Заказ Х-1040.
Отпечатано в типографии ООО «Экопейпер», 420044, Россия, г. Казань, пр. Ямашева, д. 36Б.

Краткое содержание

https://t.me/it_boooks/2

Предисловие 16

Благодарности 21

О книге 22

Об авторе 27

Иллюстрация на обложке 28

Часть I. Примитивы: криптографические ингредиенты

Глава 1. Введение 31

Глава 2. Хеш-функции 55

Глава 3. Имитовставки 80

Глава 4. Аутентифицированное шифрование 98

Глава 5. Обмен ключами 124

Глава 6. Асимметричный и гибридный виды шифрования 148

Глава 7. Подписи и доказательства с нулевым разглашением 171

Глава 8. Случайность и секреты 196

Часть II. Протоколы: рецепты применения криптографии

Глава 9. Безопасная передача данных 223

Глава 10. Сквозное шифрование 251

Глава 11. Аутентификация пользователей 280

Глава 12. Криптография и криптовалюты 309

Глава 13. Аппаратная криптография 338

Глава 14. Постквантовая криптография 362

Глава 15. Криптография следующего поколения 389

Глава 16. Условия, в которых криптография может не сработать 415

Приложение. Ответы к упражнениям 429

Оглавление

Предисловие 16
Книга, которая создавалась много лет 16
Реальный учебный план для криптографов 17
Где находится большинство ошибок 18
Нужно ли было писать еще одну книгу? 19

Благодарности 21

О книге 22
Кому следует прочитать книгу 22
Студенты 22
Практикующие специалисты 22
Разработчики, прямо или косвенно использующие криптографию 23
Криптографы, интересующиеся другими областями 23
Инженеры и менеджеры по продуктам, желающие лучше разобраться в теме 23
Люди, которые хотят узнать о криптографии, применяемой в реальной жизни 23
Уровень подготовки читателя 24
Как организована книга 24
Примеры исходного кода 26
От издательства 26

Об авторе 27

Иллюстрация на обложке 28

Часть I. Примитивы: криптографические ингредиенты

Глава 1. Введение 31
1.1. Криптография как защита протоколов 32
1.2. Симметричная криптография. Что такое симметричное шифрование . . . 33
1.3. Принцип Керкгоффса: секретом является только ключ. 35
1.4. Асимметричная криптография: два ключа лучше одного. 38
1.4.1. Обмен ключами, или Как получить общий секрет 38
1.4.2. Асимметричное шифрование, не похожее на симметричное 40
1.4.3. Цифровые подписи, прямо как подписи на бумаге 43
1.5. Классификация и обобщение в области криптографии 45
1.6. Криптография в теории и криптография реального мира 47
1.7. От теории к практике: выбери себе приключение 48
1.8. Предупреждение 53
Резюме 54

Глава 2. Хеш-функции 55
2.1. Что такое хеш-функция 55
2.2. Свойства безопасности хеш-функции 58
2.3. Методы обеспечения безопасности для хеш-функций 61
2.4. Хеш-функции на практике 62
2.4.1. Обязательства 63
2.4.2. Целостность субресурсов 63
2.4.3. BitTorrent 64
2.4.4. Tor 64
2.5. Стандартизированные хеш-функции 65
2.5.1. Хеш-функция SHA-2 66
2.5.2. Хеш-функция SHA-3 69
2.5.3. SHAKE и cSHAKE — две функции расширенного выхода (XOF) . . 73
2.5.4. Избегание неоднозначного хеширования с помощью TupleHash 75
2.6. Хеширование паролей 77
Резюме 79

Глава 3. Имитовставки 80
3.1. Файлы cookie без сохранения состояния: поясняющий пример имитовставок 81
3.2. Пример с кодом 84
3.3. Свойства безопасности имитовставки 85
3.3.1. Подделка аутентификационного тега 85
3.3.2. Длина аутентификационного тега 86
3.3.3. Атаки повторного воспроизведения 87
3.3.4. Проверка аутентификационных тегов за постоянное время 88
3.4. Имитовставка в реальном мире 90
3.4.1. Аутентификация сообщения 90
3.4.2. Получение ключей 90

3.4.3. Целостность файлов cookie . . . 91
3.4.4. Хеш-таблицы . . . 91
3.5. Практическое использование имитовставок . . . 91
3.5.1. HMAC — имитовставка с использованием хеш-функции . . . 92
3.5.2. KMAC — имитовставка с использованием cSHAKE . . . 93
3.6. SHA-2 и атаки удлинением сообщения . . . 94
Резюме . . . 97

Глава 4. Аутентифицированное шифрование . . . 98
4.1. Что такое шифр . . . 99
4.2. Блочный шифр Advanced Encryption Standard . . . 101
4.2.1. Какой уровень криптостойкости обеспечивает AES . . . 101
4.2.2. Интерфейс AES . . . 102
4.2.3. Внутренняя структура AES . . . 103
4.3. Зашифрованный пингвин и режим сцепления блоков шифротекста . . . 105
4.4. Отсутствие подлинности и решение проблемы в виде AES-CBC-HMAC . . . 108
4.5. Комплексные конструкции. Аутентифицированное шифрование . . . 110
4.5.1. Что такое аутентифицированное шифрование с присоединенными данными . . . 110
4.5.2. AEAD-алгоритм AES-GCM . . . 112
4.5.3. ChaCha20Poly1305 . . . 117
4.6. Другие виды симметричного шифрования . . . 121
4.6.1. Обертка ключа . . . 121
4.6.2. Аутентифицированное шифрование, устойчивое к некорректному применению nonce . . . 121
4.6.3. Шифрование диска . . . 122
4.6.4. Шифрование базы данных . . . 122
Резюме . . . 123

Глава 5. Обмен ключами . . . 124
5.1. Что такое обмен ключами . . . 125
5.2. Алгоритм обмена ключами Диффи — Хеллмана . . . 128
5.2.1. Теория групп . . . 128
5.2.2. Задача дискретного логарифма — основа алгоритма Диффи — Хеллмана . . . 132
5.2.3. Стандарты, описывающие алгоритм Диффи — Хеллмана . . . 134
5.3. Алгоритм обмена ключами Диффи — Хеллмана на эллиптических кривых . . . 136
5.3.1. Что такое эллиптические кривые . . . 136
5.3.2. Как работает протокол обмена ключами Диффи — Хеллмана на эллиптических кривых . . . 140
5.3.3. Стандарты, описывающие алгоритм Диффи — Хеллмана на эллиптических кривых . . . 142
5.4. Атаки через малые подгруппы и другие аспекты безопасности . . . 143
Резюме . . . 147

Глава 6. Асимметричный и гибридный виды шифрования 148
6.1. Что такое асимметричное шифрование 149
6.2. Практическое применение асимметричного шифрования и гибридное шифрование 151
6.2.1. Обмен ключами и инкапсуляция ключа 151
6.2.2. Гибридное шифрование 152
6.3. Асимметричное шифрование посредством алгоритма RSA: большее и меньшее зло 157
6.3.1. Классический алгоритм RSA 157
6.3.2. Почему не стоит использовать RSA PKCS#1 v1.5 162
6.3.3. Асимметричное шифрование с помощью RSA-OAEP 164
6.4. Гибридное шифрование с помощью алгоритма ECIES 167
Резюме 170

Глава 7. Подписи и доказательства с нулевым разглашением 171
7.1. Что такое подпись 172
7.1.1. Как на практике происходит добавление и проверка подписи 173
7.1.2. Основной пример применения подписей — аутентифицированный обмен ключами 174
7.1.3. Использование подписей в реальном мире. Инфраструктура открытых ключей 175
7.2. Доказательства с нулевым разглашением. Происхождение подписей . . . 176
7.2.1. Протокол идентификации Шнорра — интерактивное доказательство с нулевым разглашением 177
7.2.2. Подписи как неинтерактивные доказательства с нулевым разглашением 180
7.3. Алгоритмы подписи, которые следует (или не стоит) использовать 181
7.3.1. Плохой стандарт RSA PKCS#1 v1.5 182
7.3.2. Лучший стандарт RSA-PSS 185
7.3.3. Алгоритм цифровой подписи на эллиптических кривых 186
7.3.4. Алгоритм цифровой подписи на кривой Эдвардса 188
7.4. Тонкости схем подписи 192
7.4.1. Атаки на подписи через подмену 192
7.4.2. Деформируемость подписи 194
Резюме 194

Глава 8. Случайность и секреты 196
8.1. Что такое случайность 197
8.2. Медленная генерация случайности? На помощь придет генератор псевдослучайных чисел 199
8.3. Генерация случайности на практике 203
8.4. Генерация случайности и требования обеспечения безопасности 205
8.5. Открытая случайность 208
8.6. Формирование ключей с помощью HKDF 210
8.7. Управление ключами и секретами 214
8.8. Децентрализация доверия с помощью пороговой криптографии 216
Резюме 220

Часть II. Протоколы: рецепты применения криптографии

Глава 9. Безопасная передача данных . . . 223
9.1. Протоколы SSL и TLS . . . 223
9.1.1. От SSL к TLS . . . 224
9.1.2. Практическое применение протокола TLS . . . 225
9.2. Принцип работы протокола TLS . . . 227
9.2.1. TLS-рукопожатие . . . 228
9.2.2. Шифрование данных приложения с помощью протокола TLS 1.3 . . . 242
9.3. Современное состояние зашифрованного веба . . . 243
9.4. Другие безопасные транспортные протоколы . . . 246
9.5. Протокольный фреймворк Noise — современная альтернатива TLS . . . 247
9.5.1. Схемы рукопожатия фреймворка Noise . . . 247
9.5.2. Процесс рукопожатия при использовании фреймворка Noise . . . 248
Резюме . . . 250

Глава 10. Сквозное шифрование . . . 251
10.1. Зачем нужно сквозное шифрование . . . 252
10.2. Корня доверия нигде не найти . . . 254
10.3. Провалы в сфере шифрования электронной почты . . . 255
10.3.1. PGP или GPG? И как все это работает . . . 255
10.3.2. Масштабирование доверия между пользователями с помощью сети доверия . . . 259
10.3.3. Обнаружение ключей — это настоящая проблема . . . 260
10.3.4. Если не PGP, то что? . . . 262
10.4. Безопасный обмен сообщениями. Современный взгляд на сквозное шифрование на примере протокола Signal . . . 263
10.4.1. Большее удобство для пользователя по сравнению с WOT: доверяй, но проверяй . . . 265
10.4.2. X3DH — рукопожатие в рамках протокола Signal . . . 267
10.4.3. Двойной храповик — протокол Signal, используемый после рукопожатия . . . 271
10.5. Текущее состояние области сквозного шифрования . . . 276
Резюме . . . 279

Глава 11. Аутентификация пользователей . . . 280
11.1. Краткий обзор темы аутентификации . . . 281
11.2. Аутентификация пользователей, или Попытка избавиться от паролей . . . 283
11.2.1. Один пароль для всего. Технология единого входа и менеджеры паролей . . . 285
11.2.2. Не хотите видеть их пароли? Используйте асимметричный обмен ключами с парольной аутентификацией . . . 287
11.2.3. Одноразовые пароли — это не совсем пароли: беспарольная аутентификация с симметричными ключами . . . 291
11.2.4. Замена паролей асимметричными ключами . . . 295

11.3. Аутентификация, управляемая пользователем, — сопряжение устройств при участии человека . . . 298
11.3.1. Предварительно распределенные ключи . . . 300
11.3.2. Симметричный обмен ключами с парольной аутентификацией с протоколом CPace . . . 302
11.3.3. Чтобы проверить, не подвергся ли ваш обмен ключами атаке посредника, используйте короткую аутентификационную строку . . . 303
Резюме . . . 307

Глава 12. Криптография и криптовалюты . . . 309
12.1. Знакомство с византийскими отказоустойчивыми алгоритмами консенсуса . . . 310
12.1.1. Проблема устойчивости и распределенные протоколы . . . 310
12.1.2. Децентрализация помогает решить проблему доверия . . . 312
12.1.3. Проблема масштаба. Общедоступные и устойчивые к цензуре сети . . . 313
12.2. Принцип работы системы Bitcoin . . . 315
12.2.1. Обработка пользовательских балансов и транзакций в системе Bitcoin . . . 316
12.2.2. Добыча BTC в эпоху цифрового золота . . . 318
12.2.3. Форки. Разрешение конфликтов при добыче криптовалюты . . . 321
12.2.4. Уменьшение размера блока с помощью деревьев Меркла . . . 324
12.3. Экскурсия по миру криптовалют . . . 326
12.3.1. Волатильность . . . 326
12.3.2. Время ожидания . . . 327
12.3.3. Размер блокчейна . . . 327
12.3.4. Конфиденциальность . . . 328
12.3.5. Энергоэффективность . . . 328
12.4. DiemBFT — византийский отказоустойчивый протокол консенсуса . . . 328
12.4.1. Безопасность и живость — два свойства протокола консенсуса BFT . . . 329
12.4.2. Раунд в протоколе DiemBFT . . . 329
12.4.3. Какое количество нечестных участников допускает протокол . . . 331
12.4.4. Правила голосования в DiemBFT . . . 331
12.4.5. Когда транзакции считаются завершенными . . . 332
12.4.6. Интуитивное понимание безопасности протокола DiemBFT . . . 333
Резюме . . . 336

Глава 13. Аппаратная криптография . . . 338
13.1. Модель злоумышленника в современной криптографии . . . 338
13.2. Недоверенные среды — аппаратное обеспечение спешит на помощь . . . 340
13.2.1. Криптография белого ящика — это плохая идея . . . 341
13.2.2. Они в вашем бумажнике: смарт-карты и защищенные элементы . . . 342
13.2.3. Излюбленные банками аппаратные модули безопасности . . . 345

13.2.4. Доверенные платформенные модули — полезная стандартизация защищенных элементов . . . 347
13.2.5. Конфиденциальные вычисления с помощью доверенной среды выполнения . . . 351
13.3. Какое решение подходит именно мне? . . . 352
13.4. Криптография, устойчивая к утечкам, или Как смягчить атаки по сторонним каналам в программном обеспечении . . . 354
13.4.1. Программирование операций, выполняемых за постоянное время . . . 357
13.4.2. Не используйте секрет! Маскировка и ослепление . . . 358
13.4.3. А как насчет атак по ошибкам вычислений? . . . 359
Резюме . . . 360

Глава 14. Постквантовая криптография . . . 362
14.1. Что такое квантовые компьютеры и почему они пугают криптографов . . 363
14.1.1. Квантовая механика, изучение мельчайших частиц . . . 363
14.1.2. От создания квантовых компьютеров до квантового превосходства . . . 366
14.1.3. Влияние алгоритмов Гровера и Шора на криптографию . . . 367
14.1.4. Постквантовая криптография, защита от квантовых компьютеров . . . 369
14.2. Подписи на основе хеша: не нужно ничего, кроме хеш-функции . . . 369
14.2.1. Одноразовые подписи с подписями Лэмпорта . . . 370
14.2.2. Ключи меньшего размера с одноразовыми подписями Винтерница . . . 372
14.2.3. Многоразовые подписи с XMSS и SPHINCS+ . . . 374
14.3. Сокращение длины ключей и подписей с помощью криптографии на решетках . . . 378
14.3.1. Что такое решетка . . . 378
14.3.2. Обучение с ошибками — основа для криптографических систем? . 380
14.3.3. Kyber — обмен ключами на основе решеток . . . 381
14.3.4. Dilithium — схема цифровой подписи на основе решетки . . . 384
14.4. Стоит ли паниковать . . . 386
Резюме . . . 387

Глава 15. Криптография следующего поколения . . . 389
15.1. Чем больше народу, тем веселее: защищенные многосторонние вычисления . . . 390
15.1.1. Пересечение закрытых множеств . . . 391
15.1.2. MPC общего назначения . . . 392
15.1.3. Текущее состояние дел в сфере MPC . . . 395
15.2. Полностью гомоморфное шифрование и обещание зашифрованного облака . . . 395
15.2.1. Пример гомоморфного шифрования с помощью алгоритма RSA . . 396
15.2.2. Различные типы гомоморфного шифрования . . . 397

15.2.3. Перешифрование — ключ к полностью гомоморфному шифрованию . . . 397
15.2.4. Схема полностью гомоморфного шифрования, основанная на задаче обучения с ошибками . . . 401
15.2.5. Сфера применения полностью гомоморфного шифрования . . . 402
15.3. Многоцелевые доказательства с нулевым разглашением . . . 403
15.3.1. Принцип работы схем zk-SNARK . . . 406
15.3.2. Гомоморфные обязательства для сокрытия частей доказательства . . . 407
15.3.3. Билинейные спаривания для улучшения гомоморфных обязательств . . . 407
15.3.4. Откуда берется лаконичность . . . 408
15.3.5. От программ к многочленам . . . 409
15.3.6. Программы предназначены для компьютеров, а нам нужны арифметические схемы . . . 410
15.3.7. От арифметической схемы к системе ограничений ранга 1 . . . 410
15.3.8. От системы ограничений ранга 1 к многочлену . . . 411
15.3.9. Для вычисления значения многочлена, скрытого в экспоненте, нужны двое . . . 412
Резюме . . . 414

Глава 16. Условия, в которых криптография может не сработать . . . 415
16.1. Поиск подходящего криптографического примитива или протокола — скучное занятие . . . 416
16.2. Как использовать криптографический примитив или протокол. Вежливые стандарты и формальная верификация . . . 417
16.3. Где искать хорошие библиотеки . . . 420
16.4. Злоупотребление криптографией. Разработчики — главные враги . . . 422
16.5. Вы делаете это неправильно: удобная безопасность . . . 423
16.6. Криптография — это не изолированный остров . . . 425
16.7. Ваша обязанность как практикующего криптографа состоит в том, чтобы не развертывать собственную криптографию . . . 425
Резюме . . . 427

Приложение. Ответы к упражнениям . . . 429

Моим родителям Анне Серкле и Генри Вонгу,
которые воспитали во мне любознательность.

Моей жене Фелиции Лупу, которая поддерживала
меня на протяжении всего этого путешествия.

Предисловие

Взяв в руки эту книгу, вы, вероятно, задались вопросом: зачем понадобилась еще одна книга по криптографии? Или почему вам стоит ее прочитать? Чтобы ответить на этот вопрос, мне следует объяснить, с чего все началось.

КНИГА, КОТОРАЯ СОЗДАВАЛАСЬ МНОГО ЛЕТ

Сегодня, если хотите о чем-либо узнать, вы заходите в Google, Bing или Baidu. Однако, если речь идет о криптографии, этих ресурсов может оказаться недостаточно. Я столкнулся с этой проблемой очень давно, и она до сих пор напрягает меня.

Однажды, когда я еще учился, одним из моих заданий была реализация дифференциальной атаки по энергопотреблению, которая в то время была настоящим прорывом в криптоанализе и первой из опубликованных атак по сторонним каналам. Дифференциальная атака по энергопотреблению представляет собой нечто поистине волшебное, так как позволяет выведать секреты устройства путем измерения уровня его энергопотребления в процессе шифрования или расшифровки данных. Ознакомившись с научными статьями на эту тему, я пришел к выводу, что изложение идей в них не отличается ясностью и доходчивостью. Я помню, как бился головой о стену, пытаясь понять, что именно хотели сказать авторы. А еще хуже было то, что я не мог найти интернет-ресурсы, которые объяснили бы суть той или иной статьи. Так что мне пришлось доходить до всего своим умом. И тогда я подумал, что, может быть, смогу помочь другим людям, столкнувшимся с той же проблемой.

Воодушевленный этой идеей, я нарисовал несколько диаграмм, анимировал их и сопроводил устными комментариями. Это было мое первое видео на YouTube, посвященное криптографии: https://www.youtube.com/watch?v=gbqNCgVcXsM.

Спустя годы после публикации этого видео я все еще получаю благодарности от незнакомых пользователей. Только вчера, пока я писал это предисловие, кто-то оставил под видео комментарий: «Спасибо, это действительно отличное объяснение сути статьи, которое сэкономило мне много часов».

Вот это награда! Этого маленького шага было достаточно для того, чтобы я захотел сделать еще больше. Я начал все чаще записывать подобные видео, а затем завел блог, посвященный криптографии. Вы можете найти его по адресу https://cryptologie.net.

Прежде чем приступить к написанию этой книги, я собрал около 500 статей, объясняющих множество концепций. Все это было лишь практикой. А сама идея написания книги возникла у меня за несколько лет до того, как издательство Manning обратилось ко мне с соответствующим предложением.

РЕАЛЬНЫЙ УЧЕБНЫЙ ПЛАН ДЛЯ КРИПТОГРАФОВ

Получив степень бакалавра по фундаментальной математике, я не знал, что делать дальше. Кроме того, я всю жизнь занимался программированием и хотел как-то совместить эти два направления. Естественно, я заинтересовался криптографией, которая, как мне казалось, сочетала в себе лучшее из двух миров, и начал читать книги, которые были в моем распоряжении. Вскоре я понял, что нашел свое призвание.

Правда, кое-что в этих книгах меня раздражало, в частности длинные вступления, посвященные истории вопроса. Меня всегда интересовали только технические аспекты, поэтому я дал себе слово, что в случае написания собственной книги по криптографии я не посвящу ни строчки шифрам Виженера, шифрам Цезаря и прочим пережиткам прошлого. И вот, получив степень магистра по криптографии в Университете Бордо, я решил, что готов к реальной работе. Но даже не подозревал, с чем мне предстоит столкнуться.

Я считал, что моей степени будет достаточно, но оказалось, что я многого не знаю о реальных протоколах, с которыми придется иметь дело. Я потратил много времени на изучение математики на эллиптических кривых, но ничего не знал о способах их применения в криптографических алгоритмах. В ходе обучения я познакомился с LFSR, ElGamal, DES и рядом других криптографических примитивов, о которых больше никогда не услышу.

Когда я начал работать в компании Matasano, которая впоследствии была приобретена NCC Group, моим первым заданием был аудит OpenSSL — самой популярной реализации SSL/TLS-кода, с помощью которого был зашифрован весь Интернет. Как же я намучился! Помню, как каждый вечер возвращался домой с сильной головной болью. Эти библиотека и протокол были настоящей катастрофой! Тогда я даже не подозревал, что спустя годы стану одним из разработчиков последней версии этого протокола, TLS 1.3.

В тот момент я думал: «Вот чему нас должны были учить. Знания, которые я получаю сейчас, позволили бы мне подготовиться к реальной жизни!» Ведь теперь я был практиком в области безопасности, специализирующимся на криптографии. Я изучал реальные криптографические приложения и выполнял ту работу, о которой мечтают обладатели ученой степени по криптографии. Я реализовывал, проверял, использовал и давал советы относительно того, какие криптографические алгоритмы следует применять в той или иной ситуации. Именно поэтому я самый первый читатель книги, которую пишу. В ней содержится то, что стоило бы знать моему прошлому «я», чтобы как следует подготовиться к реальной работе.

ГДЕ НАХОДИТСЯ БОЛЬШИНСТВО ОШИБОК

В ходе своей консалтинговой практики мне довелось провести аудит множества реальных криптографических приложений, таких как OpenSSL, система создания зашифрованных резервных копий Google, реализация TLS 1.3 от Cloudflare, протокол центра сертификации Let's Encrypt, протокол Sapling для криптовалюты Zcash, пороговая схема повторного прокси-шифрования NuCypher и десятки других реальных криптографических приложений, о которых я, к сожалению, не могу говорить публично.

В самом начале моей деятельности мне поручили провести аудит протокола, разработанного известной корпорацией для шифрования своих сообщений. Оказалось, что он подписывал практически все, кроме эфемерных ключей, которые можно было легко заменить, сломав весь протокол. Это была типичная ошибка новичка, но люди, считавшие себя достаточно опытными для того, чтобы создать собственную криптосистему, упустили ее из виду. Помню, как после моего объяснения этой уязвимости все находившиеся в комнате инженеры на целых 30 секунд погрузились в молчание.

С тех пор подобные истории повторялись много раз. Однажды, проводя аудит криптовалюты для другого клиента, я выявил возможность подделки уже существующих транзакций, обусловленную некоторой двусмысленностью подписываемых данных. Изучая реализацию TLS по заказу еще одного клиента, я нашел несколько способов взлома алгоритма RSA, что впоследствии привело к написанию совместно с одним из разработчиков RSA технического документа, способствовавшего выявлению ряда общеизвестных уязвимостей (CVE) в десятке проектов с открытым исходным кодом. Совсем недавно, уже в ходе работы над книгой, я изучил новый протокол Matrix, применяемый для шифрования переписки в чате, и обнаружил недостатки в его протоколе аутентификации, нарушавшие процесс сквозного шифрования. При использовании криптографии очень многое может пойти не так. В какой-то момент я понял, что должен написать об этом. Именно поэтому в моей книге вы найдете множество примеров из практики.

В рамках своей работы я изучал криптографические библиотеки и приложения, написанные на множестве языков программирования. Я выявлял ошибки (в частности, CVE-2016-3959 в стандартной библиотеке Golang), исследовал способы, с помощью которых библиотеки могут ввести вас в заблуждение (см., например, мою статью *How to Backdoor Diffie — Hellman*), и давал советы разработчикам относительно того, какие библиотеки им следует использовать в тех или иных случаях.

В дальнейшем я разработал протокол Disco (https://discocrypto.com, https://embeddeddisco.com) и написал его полнофункциональную криптографическую библиотеку, состоящую менее чем из 1000 строк кода, причем на нескольких языках. Протокол Disco опирался всего на два криптографических примитива: пермутацию SHA-3 и эллиптическую кривую Curve25519. Используя только эти две вещи, реализованные с помощью 1000 строк кода, разработчик мог выполнить любой тип аутентифицированного обмена ключами, шифрования, хеширования, формирования ключей, создания цифровых подписей, имитовставок и т. д.

Это дало мне уникальную возможность понять, какой должна быть хорошая криптографическая библиотека.

Я хотел, чтобы моя книга содержала именно такие практические идеи, поэтому включил в нее примеры применения криптографии на разных языках программирования с использованием зарекомендовавших себя криптографических библиотек.

НУЖНО ЛИ БЫЛО ПИСАТЬ ЕЩЕ ОДНУ КНИГУ?

Когда я проводил один из ежегодных тренингов по криптографии в рамках известной конференции по информационной безопасности Black Hat, ко мне подошел один студент и попросил порекомендовать хорошую книгу или онлайн-курс по этой теме. Я посоветовал прочитать книгу Боне и Шоупа и пройти курс Боне *Cryptography I на Coursera*. (Вы найдете соответствующие рекомендации в конце этой книги.)

На это студент сказал: «Я пробовал, там слишком много теории!» Я запомнил его ответ. Сначала я с ним не согласился, но постепенно осознал его правоту. Большинство ресурсов перегружено математикой, с которой разработчики, занимающиеся криптографией, обычно не хотят иметь дела. Что же им остается?

Двумя другими более или менее авторитетными ресурсами в то время были книги Брюса Шнайера *Applied Cryptography*[1] и *Cryptography Engineering*. Но они уже начали устаревать. В «Прикладной криптографии» четыре главы были посвящены блочным шифрам, целая глава — режимам их работы, а аутентифицированному шифрованию — ни одной. В самом последнем издании *Cryptography Engineering* криптография на эллиптических кривых упоминалась лишь в сноске. В то же время многие из моих видеороликов и статей в блоге превращались в полезные ресурсы, объясняющие некоторые криптографические концепции. Я знал, что могу сделать что-то значимое.

Постепенно многие мои студенты начали интересоваться криптовалютами и задавать все больше вопросов на эту тему. А я все чаще занимался аудитом криптовалютных приложений. Позднее я устроился в Facebook[2], где возглавил работу по обеспечению безопасности криптовалюты Libra (ныне известна как Diem). В то время криптовалюты были одной из самых быстро развивающихся сфер деятельности, в которой применялось множество чрезвычайно интересных криптографических примитивов, ранее практически не использовавшихся в реальном мире (в частности, доказательства с нулевым разглашением, агрегированные подписи, пороговая криптография, многосторонние вычисления, протоколы консенсуса, криптографические аккумуляторы, верифицируемые случайные функции, верифицируемые функции задержки и т. д.). И все же ни в одной книге по криптографии не было главы, посвященной криптовалютам. Я оказался в уникальном положении.

---

[1] *Шнайер Б.* Прикладная криптография: протоколы, алгоритмы и исходный код на С. 2-е изд. — 2022.

[2] Компания Meta, которой принадлежит Facebook, запрещена в РФ. — *Примеч. ред.*

Я знал, что могу рассказать студентам, разработчикам, консультантам, инженерам по информационной безопасности и другим людям о том, что представляет собой современная прикладная криптография. И решил написать книгу, содержащую как можно меньше формул и исторических отсылок и как можно больше диаграмм и примеров неудачного применения криптографии, которые я наблюдал собственными глазами. Эта книга должна была фокусироваться не на устаревших алгоритмах, а на таких широко используемых криптографических концепциях, как протоколы TLS и Signal, протокольный фреймворк Noise, криптовалюты, аппаратные модули безопасности, пороговая криптография и т. д. Она должна была содержать как можно меньше теории и как можно больше информации по таким актуальным темам, как обмен ключами с парольной аутентификацией, доказательства с нулевым разглашением, постквантовая криптография и т. п.

Когда в 2018 году издательство Manning обратилось ко мне с вопросом, не хочу ли я написать книгу о криптографии, я уже знал ответ. И знал, о чем хочу написать. Я просто ждал, пока кто-нибудь даст мне возможность и повод потратить время на написание книги, которую давно задумал. Кроме того, она отлично вписывалась в одну из серий издательства Manning. Итак, книга, которую вы держите в руках, является результатом более чем двух лет упорного труда и огромной любви. Надеюсь, она вам понравится.

Благодарности

Я благодарю Марину Майклс за постоянную помощь и ценные идеи, без которых эта книга, возможно, не была бы дописана.

Хочу также сказать спасибо Фрэнсису Бурану, Сэму Зайделю, Майклу Розенбергу, Паскалю Кнехту, Сету Дэвиду Шоэну, Эялю Ронену, Саралинн Чик, Роберту Сикорду, Элою Мануэлю, Робу Вуду, Хантеру Монку, Жану-Кристофу Форесту, Ливиу Барте, Маттиа Реджани, Оливье Герре, Андрею Лабунову, Карлу Литтке, Яну Ивницкому, Келлеру Фуксу, Роману Забицки, М. К. Сараванану, Саре Зенну, Даниэлю Бурдрезу, Джейсону Ноллу, Илиасу Черкауи, Фелипе де Лима, Раулю Силесу, Маттео Бокки, Джону Вудсу, Костасу Халкиасу, Йолану Ромайеру, Херардо ди Джакомо, Грегори Назарио, Робу Стаббсу, Яну Янчару, Гейбу Пайку, Кирану Туммале, Стивену Сингаму, Джереми О'Донохью, Джереми Буну, Томасу Дюбуше, Шарлю Гиллеме, Райану Сливи, Лионелю Ривьере, Бенджамину Ларсену, Габриэлю Жионо, Даану Шпренкельсу, Андреасу Крогену, Вадиму Любашевскому, Самуэлю Невесу, Стивену (Донгзе) Юэ, Тони Патти, Грэму Стилу и всем комментаторам с форума liveBook за многочисленные дискуссии и исправления, а также технические и редакторские замечания.

Кроме того, выражаю благодарность всем рецензентам: Адхиру Рамджиавану, Элу Пежевски, Алю Рахими, Алессандро Кампеису, Бобби Лину, Чаду Дэвису, Дэвиду Т. Кернсу, Доминго Салазару, Эдди Влуггену, Габору Ласло Хайбе, Геерту ван Лайтему, Гжегожу Бернасу, Харальду Куну, Уго Дурану, Яну Питеру Хервейеру, Джеффу Смиту, Джиму Карабатсосу, Джоэлу Котарски, Джону Параскевопулосу, Мэтту ван Винклу, Михалу Рутке, Полу Гребенку, Ричарду Лебелю, Руслану Шевченко, Сандживу Джайсвалу, Шону П. Болану, Томасу Дойленду, Уильяму Руденмальму, чьи предложения помогли существенно улучшить книгу.

О книге

С тех пор как я приступил к написанию книги «Реальная криптография», прошло уже более двух лет. Изначально я планировал просто рассказать в ней обо всех видах криптографии, которые существуют в реальном мире. Но, разумеется, это невыполнимая задача. Ни одну область нельзя обобщить в одной-единственной книге. Поэтому мне пришлось искать оптимальный баланс между количеством сообщаемых читателю деталей и размером области, которую хотелось охватить. Надеюсь, мне это удалось. Если вы ищете практическую книгу, рассказывающую о принципах работы реальной криптографии, которая используется компаниями и внедряется в продукты, а не справочник, содержащий все детали реализации, то эта книга для вас.

КОМУ СЛЕДУЕТ ПРОЧИТАТЬ КНИГУ

Далее я перечислю категории людей, которым, по моему мнению, будет полезна эта книга.

Студенты

Если вы изучаете информатику, информационную безопасность (ИБ) или криптографию и хотите узнать о том, как криптография применяется в реальном мире (потому что собираетесь работать в этой отрасли или заниматься научными исследованиями), то этот учебник для вас. Почему? Потому что, как сказано в предисловии, я и сам был студентом и написал такую книгу, которую хотел бы иметь под рукой во время учебы.

Практикующие специалисты

Пентестеры, ИБ-консультанты, ИБ-инженеры, ИБ-архитекторы и прочие специалисты по информационной безопасности составляли большинство моих студентов, когда я преподавал прикладную криптографию. В связи с этим материал книги был доработан с учетом многочисленных вопросов, которые мне задавали в процессе объяснения сложных криптографических концепций далеким от криптографии людям. Поскольку я и сам являюсь практиком в сфере информационной

безопасности, то включил в эту книгу сведения о криптографических инструментах, которые анализировал по заказу крупных компаний, и об ошибках, с которыми столкнулся в ходе этого.

Разработчики, прямо или косвенно использующие криптографию

На материал данной книги повлияли также многочисленные дискуссии с клиентами и коллегами, которые в большинстве своем не являются ни практикующими ИБ-специалистами, ни криптографами. Сегодня становится все труднее писать код, не задействующий тот или иной криптографический инструмент, поэтому разработчикам необходимо иметь некоторое представление о том, что они используют. Данная книга поможет заинтересованным читателям разобраться в работе разных инструментов с помощью примеров кода, написанных на различных языках программирования, и многого другого.

Криптографы, интересующиеся другими областями

Эта книга представляет собой введение в прикладную криптографию, которое может оказаться полезным для людей вроде меня. Как вы помните, я писал ее в первую очередь для себя. Если мне удалось воплотить задуманное, то, прочитав соответствующую главу этой книги, криптограф-теоретик быстро поймет, как выглядит мир прикладной криптографии; криптограф, занимающийся симметричным шифрованием, сможет быстро разобраться в том, как происходит обмен ключами с парольной аутентификацией; криптограф, работающий с протоколами, сможет получить некоторое представление о квантовой криптографии и т. д.

Инженеры и менеджеры по продуктам, желающие лучше разобраться в теме

Эта книга призвана также ответить на продукт-ориентированные вопросы. Каковы компромиссы и ограничения тех или иных подходов? Чем я рискую? Поможет ли данный путь обеспечить соответствие нормативным требованиям? Что мне нужно сделать для того, чтобы работать с государственными структурами?

Люди, которые хотят узнать о криптографии, применяемой в реальной жизни

Чтобы прочитать эту книгу, не обязательно принадлежать к какой-то из перечисленных категорий специалистов. Если вас интересуют способы использования криптографии в реальном мире, этого будет достаточно. Однако имейте в виду, что здесь я не касаюсь истории криптографии и основ информатики, так что, прежде чем браться за чтение, не помешает получить хоть какое-то представление о криптографии.

УРОВЕНЬ ПОДГОТОВКИ ЧИТАТЕЛЯ

Чтобы извлечь максимальную пользу из чтения этой книги, вы должны иметь базовое понимание принципов работы вашего ноутбука и сети Интернет, а также хоть какое-то представление о шифровании. Книга посвящена реальной криптографии, поэтому будет сложно понять ее содержание, если вы не очень хорошо разбираетесь в компьютерах или слышите о шифровании впервые.

Если вы хоть немного представляете, во что ввязываетесь, то большим плюсом будет знание того, что такое биты и байты, и опыт использования таких побитовых операций, как XOR, сдвиг влево и т. п. Помешает ли их отсутствие понять материал книги? Нет, но вам, вероятно, придется периодически прерывать чтение, чтобы поискать в Интернете ответы на возникающие вопросы.

На самом деле вне зависимости от уровня подготовки при чтении этой книги вам, скорее всего, придется время от времени останавливаться, чтобы получить дополнительную информацию из Интернета, либо потому, что я, к своему стыду, забыл дать определение того или иного термина, прежде чем его применять, либо потому, что ошибочно предположил, что вы уже с ним знакомы. В любом случае это не должно стать большой проблемой, так как я постарался объяснить используемые концепции максимально простым языком.

Наконец, услышав слово «криптография», вы, вероятно, в первую очередь думаете о математике. Однако не стоит слишком сильно переживать по этому поводу. Книга «Реальная криптография» призвана дать вам понимание принципов работы криптографических инструментов без лишних математических подробностей.

Конечно, я бы солгал, если бы сказал, что при написании книги удалось полностью обойтись без математики. Без нее невозможно преподавать криптографию. Поэтому хороший уровень математической подготовки, несомненно, будет вам очень полезен, однако его отсутствие не должно помешать разобраться в большей части материала. При отсутствии глубокого понимания математики некоторые главы могут показаться сложными, особенно главы 14 и 15, посвященные постквантовой криптографии и криптографии следующего поколения, но вы сможете одолеть их, если проявите силу воли и поищете в Интернете информацию о матричном умножении и других неизвестных вам вещах. Если же решите пропустить эти главы, обязательно прочитайте главу 16 — это настоящая вишенка на торте.

КАК ОРГАНИЗОВАНА КНИГА

Книга «Реальная криптография» состоит из двух частей. Первую следует прочитать от начала до конца, так как она охватывает криптографические ингредиенты, которые вы будете использовать в качестве строительных блоков для создания более сложных систем и протоколов.

- Глава 1 представляет собой введение в реальную криптографию, дающее некоторое представление о том, что вы узнаете далее в книге.
- Глава 2 посвящена хеш-функциям — фундаментальному криптографическому алгоритму, используемому для создания уникальных идентификаторов из байтовых строк.
- В главе 3 рассказывается об использовании имитовставок, аутентификации данных и о том, как можно предотвратить изменение ваших сообщений третьими лицами.
- В главе 4 речь пойдет о шифровании, позволяющем двум собеседникам скрыть свои коммуникации от посторонних глаз.
- В главе 5 рассказывается об обмене ключами, который позволяет согласовывать общий секрет с другой стороной в интерактивном режиме.
- В главе 6 описывается асимметричное шифрование, которое позволяет нескольким людям шифровать сообщения, адресованные одному человеку.
- В главе 7 рассказывается о цифровых подписях — криптографических аналогах рукописной подписи, сделанной на бумаге.
- В главе 8 речь пойдет о случайности и о способах управления собственными секретами.

Часть II книги посвящена системам, состоящим из компонентов, описанных в первой части.

- В главе 9 вы узнаете об использовании шифрования и аутентификации для защиты каналов связи между машинами, в частности о протоколе SSL/TLS.
- Глава 10 посвящена сквозному шифрованию, позволяющему людям вроде нас с вами доверять друг другу.
- В главе 11 рассказывается о том, как машины аутентифицируют людей и как люди могут помочь машинам синхронизироваться друг с другом.
- Глава 12 знакомит читателей с зарождающейся криптовалютной сферой.
- Глава 13 посвящена аппаратной криптографии — устройствам, которые можно использовать для предотвращения извлечения ключей.

Далее следуют две бонусные главы: глава 14, посвященная постквантовой криптографии, и глава 15, посвященная криптографии следующего поколения. Эти два направления начинают проявляться в продуктах различных компаний, потому что постепенно становятся либо все более актуальными, либо все чаще применяемыми на практике и эффективными. Если вы пропустите эти две главы, я не стану вас осуждать, однако вам непременно следует прочитать заключительную главу, прежде чем поставить книгу на полку. В главе 16 кратко изложены различные проблемы и уроки, о которых необходимо помнить специалисту по криптографии, то есть вам. Как говорил дядя Человека-паука: «С большой силой приходит большая ответственность».

ПРИМЕРЫ ИСХОДНОГО КОДА

Книга содержит множество примеров исходного кода как в виде пронумерованных листингов, так и в виде отдельных фрагментов в обычном тексте. В обоих случаях исходный код отформатирован с помощью `вот такого моноширинного шрифта` с целью его выделения из обычного текста. Иногда измененный фрагмент кода, например дополненный новой функцией, выделяется **полужирным шрифтом**.

Во многих случаях изначальный исходный код пришлось переформатировать, например добавить символы перехода на новую строку и изменить выравнивание, чтобы разместить код на ограниченном пространстве страницы. В редких случаях даже этого оказалось недостаточно и в листинги были включены значки продолжения строки ➥. Кроме того, если код подробно описывался в тексте, то часто комментарии удалялись из соответствующего листинга. Многие листинги сопровождаются аннотациями к коду, где сделан акцент на наиболее важных концепциях.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Об авторе

Дэвид Вонг является старшим инженером по криптозащите в компании O(1) Labs, где работает над криптовалютой Mina. Ранее он возглавлял работу по обеспечению безопасности криптовалюты Diem (официально известной как Libra) в рамках проекта Novi компании Facebook, а до этого был консультантом по информационной безопасности в команде Cryptography Services компании NCC Group.

В рамках карьеры Вонг принял участие в нескольких финансируемых государством аудитах проектов с открытым исходным кодом, таких как OpenSSL и Let's Encrypt. Он выступал на различных конференциях, в том числе DEF CON и Black Hat, в рамках которой неоднократно преподавал курс по криптографии. Внес вклад в разработку стандарта TLS 1.3 и протокольного фреймворка Noise. Обнаружил уязвимости во множестве систем, включая CVE-2016-3959 в стандартной библиотеке Golang, а также CVE-2018-12404, CVE-2018-19608, CVE-2018-16868, CVE-2018-16869 и CVE-2018-16870 в различных библиотеках TLS.

Помимо прочего, Дэвид — автор протокола Disco (www.discocrypto.com и www.embeddeddisco.com) и проекта по обеспечению безопасности децентрализованных приложений для смарт-контрактов (www.dasp.co). Его исследования посвящены атакам на кэш, направленным против алгоритма RSA (http://cat.eyalro.net/), протоколу на основе QUIC (https://eprint.iacr.org/2019/028), атакам по времени, направленным против алгоритма ECDSA (https://eprint.iacr.org/2015/839), и бэкдорам в алгоритме Диффи — Хеллмана (https://eprint.iacr.org/2016/644). Более подробную информацию можно найти в его блоге по адресу www.cryptologie.net.

Иллюстрация на обложке

Иллюстрация на обложке называется «Индеец из Кито». Она взята из коллекции костюмов разных стран Жака Грассе де Сен-Совера (1775–1810) под названием *Costumes de Différents Pays*, опубликованной во Франции в 1797 году. Все иллюстрации в ней нарисованы и раскрашены вручную. Богатство коллекции Грассе де Сен-Совера напоминает нам о том, насколько разной была культура в различных городах и регионах мира всего 200 лет назад. Изолированные друг от друга, люди говорили на разных языках и диалектах. Как на улицах городов, так и в сельской местности происхождение, профессию и социальное положение любого человека можно было легко определить по его одежде.

С тех пор манера одеваться сильно изменилась, и разнообразие регионов, столь богатое в прошлом, сошло на нет. Сейчас сложно различить даже жителей разных континентов, не говоря уже о городах, регионах или странах. Возможно, мы пожертвовали культурным разнообразием в пользу жизни, более интересной в личном плане и быстро развивающейся в технологическом.

Сегодня, когда одну техническую книгу сложно отличить от другой, издательство Manning прославляет изобретательность и инициативность, свойственные компьютерному бизнесу, выпуская книги с обложками, которые демонстрируют богатое разнообразие жизни в регионах, наблюдавшееся более двух столетий назад и отраженное на рисунках Грассе де Сен-Совера.

Часть I

Примитивы: криптографические ингредиенты

Добро пожаловать в реальный мир криптографии! Книга, которую вы держите в руках (если решили приобрести печатную версию), разделена на две равные части, состоящие из восьми глав. Прочитав ее от начала до конца, вы узнаете почти все, что необходимо знать о криптографии, используемой в реальном мире — том самом, в котором вы сейчас находитесь.

Хотя главы части I лучше всего читать по порядку, в каждой из них указаны предварительные требования, необходимые для понимания материала, так что не рассматривайте это как строгое ограничение. Первые восемь глав посвящены основам — строительным блокам криптографии. В каждой из них представлен один компонент, рассматриваются его функции, принцип работы и методы применения совместно с другими элементами. Часть I призвана дать вам общее понимание каждого из криптографических инструментов, прежде чем мы перейдем к вопросам их использования в части II.

Удачи!

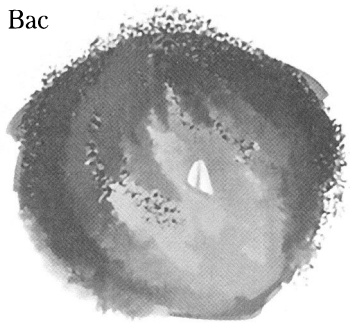
1 Введение

https://t.me/it_boooks/2

В этой главе

- ✓ В чем заключается криптография.
- ✓ Криптография в теории и криптография реального мира.
- ✓ О чем вы узнаете во время этого приключения.

Приветствую вас, путешественники. Погодите минутку. Вы собираетесь войти в мир, полный тайн и чудес, — в мир криптографии. *Криптография* — это древняя область знаний, ориентированная на защиту от темных сил. В этой книге собраны заклинания, необходимые для того, чтобы обезопасить себя от злого умысла. Многие пытались овладеть этим искусством, но лишь малая их часть выдержала испытания, возникшие на этом пути. Вас ждут захватывающие приключения!



Из этой книги мы узнаем, как криптографические алгоритмы защищают наши письма, идентифицируют союзников и защищают сокровища от врагов. Плавание в криптографическом море будет не самым спокойным, поскольку криптография — это основа всей безопасности и секретности в нашем мире и даже малейшая ошибка может обернуться смертельной опасностью.

ПРИМЕЧАНИЕ

Не останавливайтесь, если чего-то недопоняли. В конечном счете вы во всем разберетесь.

1.1. КРИПТОГРАФИЯ КАК ЗАЩИТА ПРОТОКОЛОВ

Наше путешествие начинается с введения в криптографию — науку, направленную на защиту протоколов от диверсантов. Но сначала определим, что такое *протокол*. Говоря простым языком, протоколом является список шагов, которые кто-то (один или несколько человек) должен выполнить для достижения какой-либо цели. Например, представьте ситуацию: вы хотите вздремнуть и для этого вам нужно оставить свой волшебный меч без присмотра на несколько часов. Протокол ваших действий может быть следующим.

1. Положить оружие на землю.
2. Вздремнуть под деревом.
3. Поднять оружие с земли.

Конечно, этот протокол далек от идеала, так как во время отдыха любой может украсть меч. Следовательно, криптография — это внимание к противникам, имеющим намерение обмануть вас.

В древние времена, когда правители и генералы только и занимались тем, что предавали друг друга и планировали перевороты, одной из самых больших проблем для них было найти способ *обмена конфиденциальной информацией с теми, кому они доверяли*. Отсюда родилась идея криптографии. Прошли столетия упорного труда, прежде чем криптография стала серьезной дисциплиной, каковой является и сегодня. Сейчас ее используют повсюду для обеспечения самых базовых услуг в нашем хаотичном, суровом мире.

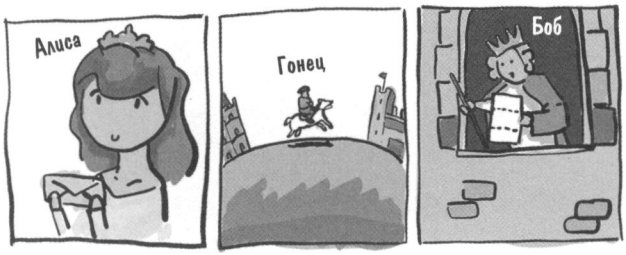
Эта книга рассказывает о практической стороне криптографии. Она возьмет вас в экспедицию по всему компьютерному миру и расскажет об используемых сегодня криптографических протоколах. Она откроет вам, из каких частей состоят протоколы и каким образом все они соединяются друг с другом. Типичная книга по данной теме начинается с рассказа об открытии криптографии и ведет читателя через всю ее историю, однако не вижу смысла выстраивать здесь такую же структуру. Я хочу рассказать о применении криптографии на практике. Хочу рассказать о том, чему сам был свидетелем, работая над криптографическими программами для крупных компаний в качестве консультанта, или о криптографии, которую сам применял, будучи инженером.

Здесь не будет почти никаких страшных математических формул. Цель книги — демистификация криптографии, исследование того, что считается полезным в настоящее время, и изложение идей о том, как устроены вещи, с которыми мы работаем. Книга предназначена для любопытных людей, заинтересованных инженеров, предприимчивых разработчиков и любознательных исследователей. С главы 1 и начинается экскурс в мир криптографии. В ней мы познакомимся с различными типами криптографии, определим их важность для нас и узнаем, как мир договорился об их использовании.

1.2. СИММЕТРИЧНАЯ КРИПТОГРАФИЯ. ЧТО ТАКОЕ СИММЕТРИЧНОЕ ШИФРОВАНИЕ

Одна из фундаментальных концепций криптографии — *симметричное шифрование*. Оно используется в большинстве криптографических алгоритмов, описанных в книге, и поэтому является чрезвычайно важным понятием. Я представлю эту новую концепцию сразу через наш первый протокол.

Представим, что королеве Алисе нужно отправить письмо лорду Бобу, между владениями которого и ее королевством стоят несколько замков. Она просит преданного гонца оседлать своего верного коня и пробиться через опасные земли, чтобы доставить драгоценное послание лорду Бобу. При этом у нее есть сомнения: гонец служит ей верой и правдой уже много лет, тем не менее она желает, чтобы передаваемое послание осталось в тайне от всех посторонних лиц, включая самого гонца! Видите ли, письмо, скорее всего, содержит скандальные сплетни о королевствах, расположенных между землями королевы Алисы и лорда Боба.



Королеве Алисе нужен протокол, имитирующий передачу сообщения лорду Бобу лично, без посредников. В практическом плане данная проблема неразрешима, если мы не введем в уравнение криптографию (или телепортацию). К этому и пришли много лет назад, изобретя новый тип криптографического алгоритма — *алгоритм симметричного шифрования*, известный также как *шифр*.

ПРИМЕЧАНИЕ

Криптографический алгоритм нередко называют *примитивом*. Примитив можно рассматривать как самую маленькую полезную криптографическую конструкцию, которая часто используется в совокупности с другими примитивами для построения протокола. Этот термин не имеет особого значения, однако он так часто встречается в литературе, что о нем полезно знать.

Давайте посмотрим, как с помощью криптографического примитива скрыть сообщение королевы Алисы от гонца. Представим, что примитив — это черный ящик (мы не можем видеть, что находится или происходит внутри) с двумя функциями:

- шифрование;
- расшифрование.

Первая функция, шифрование, принимает *секретный ключ* (обычно в виде длинного числа) и сообщение, а затем выдает серию случайных чисел, или, как говорится, некоторые зашумленные данные. Этот результат и является зашифрованным сообщением (рис. 1.1).



Рис. 1.1. Функция шифрования принимает сообщение и секретный ключ и выдает зашифрованное сообщение — длинную серию цифр, которая выглядит как хаотичный шум

Вторая функция, расшифрование, является обратной по отношению к первой. Она принимает тот же секретный ключ и случайные выходные данные первой функции (зашифрованное сообщение), а затем находит исходное сообщение (рис. 1.2).

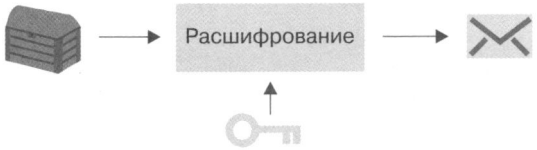


Рис. 1.2. Функция расшифрования принимает зашифрованное сообщение и секретный ключ и возвращает исходное сообщение

Прежде чем воспользоваться этим новым примитивом, королеве Алисе и лорду Бобу необходимо встретиться в реальной жизни и решить, каким будет секретный ключ. Он позволит королеве Алисе применить функцию шифрования для защиты сообщения, передать зашифрованное сообщение гонцу, который доставит его лорду Бобу. Лорд Боб, используя функцию расшифрования, с помощью того же секретного ключа сможет восстановить исходное сообщение (рис. 1.3).

При таком обмене у гонца было только то, что выглядело случайным набором символов и не давало никакого представления о содержании сообщения. По сути, благодаря криптографии мы превратили свой небезопасный протокол в безопасный. Новый протокол позволяет королеве Алисе отправить личное письмо лорду Бобу так, чтобы никто, кроме него самого, не узнал его содержания.

Использование секретного ключа для того, чтобы наполнить сообщение шумом, сделав его неотличимым от случайного набора символов, — распространенный способ защиты протокола в криптографии. В следующих главах вы увидите больше примеров его применения.

Между прочим, симметричное шифрование — это часть более широкой категории криптографических алгоритмов, называемой *симметричной криптографией* или *криптографией с секретным ключом*. Это обусловлено тем, что один и тот же ключ используют различные функции, выполняемые криптографическим примитивом. Как вы увидите позже, иногда ключей может быть несколько.

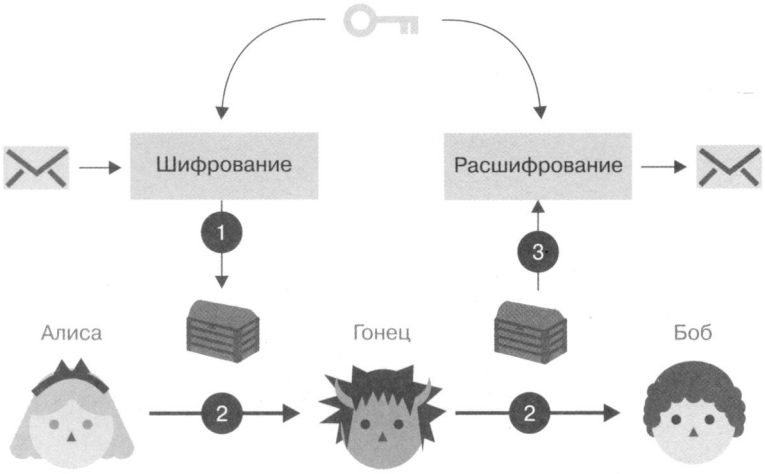


Рис. 1.3. Алиса задействует функцию шифрования с секретным ключом для преобразования своего сообщения в шум *(1)*. Она передает зашифрованное сообщение гонцу, который ничего не знает об исходном сообщении *(2)*. Получив зашифрованное сообщение, Боб может восстановить его исходное содержание с помощью функции расшифрования с тем же секретным ключом, который применила Алиса *(3)*

1.3. ПРИНЦИП КЕРКГОФФСА: СЕКРЕТОМ ЯВЛЯЕТСЯ ТОЛЬКО КЛЮЧ

Разработать криптографический алгоритм наподобие нашего криптографического примитива — просто, а вот написать *безопасный* криптографический алгоритм — задача не для слабых духом. В книге мы не станем затрагивать создание таких алгоритмов, однако *будем учиться* распознавать наиболее удачные варианты. Это не самое простое занятие, поскольку та или иная задача может иметь несколько решений. Тем не менее многочисленные неудачи в истории криптографии, а также уроки, которые сообщество извлекло из них, могут послужить хорошими подсказками. Заглянув в прошлое, мы поймем, что делает криптографический алгоритм надежным и безопасным.

Прошли сотни лет, и многие лорды и королевы покинули наш мир. С тех пор от бумаги как основного средства общения отказались в пользу более совершенных и практичных технологий. Сегодня нам доступны мощные компьютеры и Интернет. Конечно, так удобнее, но и наш подлый гонец стал намного сильнее. Он теперь повсюду: в Wi-Fi кафе Starbucks, на различных серверах, подключающих людей к Интернету и пересылающих их сообщения, и даже в машинах, на которых работают наши алгоритмы. Злоумышленники теперь могут увидеть гораздо больше сообщений, ведь каждый запрос на веб-сайт может пойти «не по тому проводу» и за считаные наносекунды его могут изменить или скопировать так, что никто этого даже не заметит.

В последние годы было много случаев, когда алгоритмы шифрования разрушались, взламывались тайными государственными организациями или независимыми исследователями, не могли защитить сообщения или соответствовать заявленным разработчиками обещаниям. Из этих неудач было извлечено много уроков, и мы постепенно пришли к пониманию того, как создать хорошую криптографическую защиту.

ПРИМЕЧАНИЕ

Криптографический алгоритм можно *взломать* разными способами. Например, возможны следующие атаки на него: секретный ключ может быть передан злоумышленнику, сообщения могут быть расшифрованы без помощи ключа, некоторые данные, касающиеся зашифрованного сообщения, могут быть раскрыты без расшифровки и т. д. Все, что каким-то образом способно снизить эффективность алгоритма, можно считать взломом.

В результате длительного процесса проб и ошибок, через который прошла криптография, сложилось устойчивое представление: получить уверенность в надежности защиты, заявленной криптографическим примитивом, можно только тогда, когда он прошел открытую проверку специалистами в этой области. В противном случае придется полагаться на *безопасность через неясность*, которая на протяжении всей истории криптографии показывала себя не с лучшей стороны. Вот почему *криптографы* (те, кто строит) обычно прибегают к помощи *криптоаналитиков* (тех, кто ломает) для проверки безопасности конструкции, хотя обычно обе функции выполняет один и тот же человек.



В качестве примера рассмотрим алгоритм шифрования Advanced Encryption Standard (AES; расширенный стандарт шифрования). Он был разработан в рамках международного конкурса, организованного Национальным институтом стандартов и технологий (National Institute of Standards and Technology, NIST).

ПРИМЕЧАНИЕ

NIST — это американское агентство, которое определяет стандарты и разрабатывает руководящие принципы в области безопасности для правительственных структур и других государственных и частных организаций. Помимо AES, агентство стандартизировало большое количество широко используемых криптографических примитивов.

Конкурс длился несколько лет, в течение которых множество криптоаналитиков-добровольцев со всего мира собирались вместе, чтобы попытаться взломать различные алгоритмы шифрования. В результате признанный на конкурсе самым

безопасным шифр и стал расширенным стандартом шифрования. Сегодня большинство людей считают AES надежным алгоритмом шифрования. Его практически всегда используют для шифрования чего-либо. Например, именно он ежедневно обеспечивает вашу защиту в Интернете.

Идея открытого создания криптографических стандартов связана с концепцией, часто называемой *принципом Керкгоффса*, который трактуется примерно так: было бы глупо полагаться на то, что наши противники не узнают, какие алгоритмы мы используем, ведь они, скорее всего, узнают о них. Давайте же обеспечивать защиту в открытую.

Если недруги королевы Алисы и лорда Боба точно знали, как те шифруют свои сообщения, то каким образом их алгоритм защищает? Ответ: *секретный ключ*! Протокол безопасен не благодаря секретности алгоритма, а благодаря секретности ключа. Вот что является общей темой этой книги: почти все криптографические алгоритмы, о которых мы узнаем и которые используются в реальном мире, можно свободно изучать и задействовать в своих проектах. Скрыты только секретные ключи, применяемые в качестве входных данных для этих алгоритмов. Как сказал Жан Робер дю Карле в 1644 году, «ars ipsi secreta magistro» («секрет в искусстве есть даже для мастера»). В следующем разделе я расскажу о совершенно ином виде криптографических примитивов. А пока упорядочим то, что мы уже знаем (рис. 1.4).

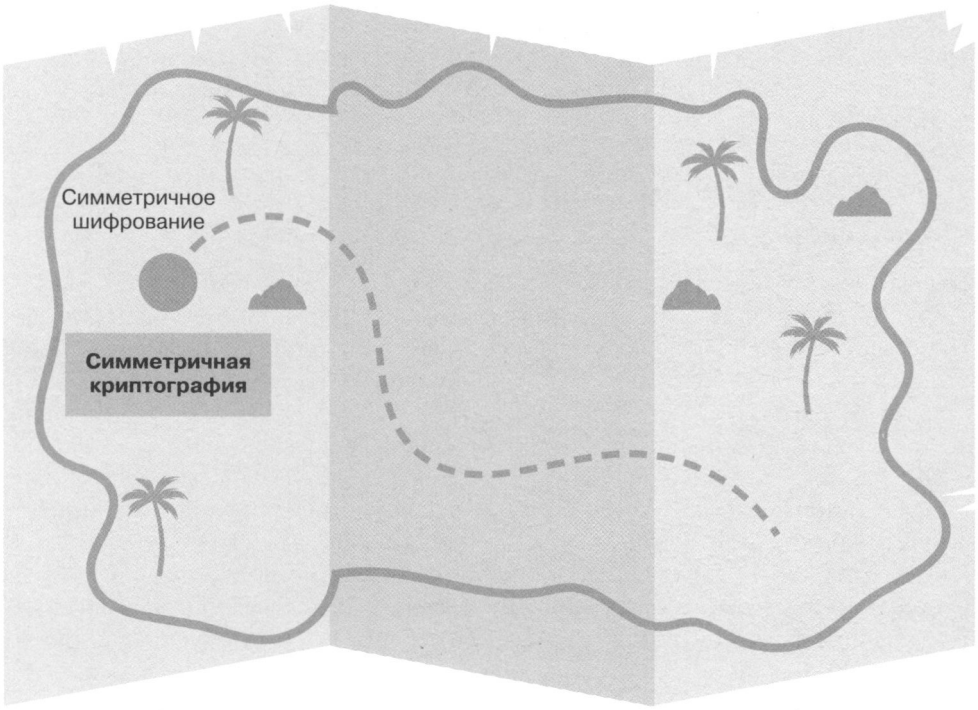


Рис. 1.4. Криптографические алгоритмы, о которых мы уже знаем.
AES — это конкретная реализация алгоритма симметричного шифрования — криптографического примитива, являющегося частью более широкого класса симметричных криптографических алгоритмов

1.4. АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ: ДВА КЛЮЧА ЛУЧШЕ ОДНОГО

В разделе про симметричное шифрование мы рассказали, как королева Алиса и лорд Боб сначала встретились и решили, каким будет симметричный ключ. Это правдоподобный сценарий, многие протоколы действительно работают именно таким образом. Но в протоколах с большим количеством участников данный способ быстро утрачивает свою практичность: нужно ли нам, чтобы наш браузер встретился с Google, Facebook, Amazon и миллиардами других веб-сайтов для обеспечения безопасного подключения?

Решение данной проблемы, которую часто обозначают как *распределение ключей*, не могли найти довольно долго — до открытия в конце 1970-х годов другой большой и полезной категории криптографических алгоритмов, именуемой *асимметричной криптографией* или *криптографией с открытым ключом*. Как правило, асимметричная криптография использует разные ключи для разных функций (в отличие от одного-единственного ключа в симметричной криптографии) или предоставляет разным участникам различный доступ к данным. В этом разделе я представлю ряд асимметричных примитивов и тем самым покажу на примерах, как криптография с открытым ключом помогает установить доверие между людьми. Обратите внимание на то, что это только краткий обзор криптографических примитивов, каждый из которых мы будем разбирать более подробно в последующих главах.

1.4.1. Обмен ключами, или Как получить общий секрет

Первый примитив асимметричной криптографии, который мы рассмотрим, — это *обмен ключами*. Названный в честь его авторов Диффи и Хеллмана (Diffie — Hellman, DH), он стал первым изобретенным и опубликованным алгоритмом с открытым ключом. Его основная задача — установление общего для двух сторон секрета, который затем может использоваться в различных целях, например в качестве ключа к примитиву симметричного шифрования.

В главе 5 я объясню работу алгоритма Диффи — Хеллмана, а сейчас ограничимся простой аналогией, чтобы понять, в чем состоит суть обмена ключами. Как и многие криптографические алгоритмы, обмен ключами обязательно начинается с того, что стороны устанавливают общий набор параметров. В нашей аналогии пусть королева Алиса и лорд Боб договорятся об использовании квадрата (■). Следующим шагом будет выбор ими личной фигуры. Оба идут в свое тайное место, и вдали от посторонних глаз королева Алиса выбирает треугольник (▲), а лорд Боб — звезду (★). Эти объекты должны оставаться в тайне любой ценой! Они представляют собой *закрытые ключи* (рис. 1.5).

После того как Алиса и Боб выбрали свои закрытые ключи, они оба по отдельности комбинируют их с общей фигурой, о которой изначально договорились, — с квадратом. В результате получаются уникальные фигуры, представляющие

собой *открытые ключи*. Теперь королева Алиса и лорд Боб могут *обмениваться* открытыми ключами (отсюда и название), поскольку те считаются публичной информацией. Данный этап изображен на рис. 1.6.

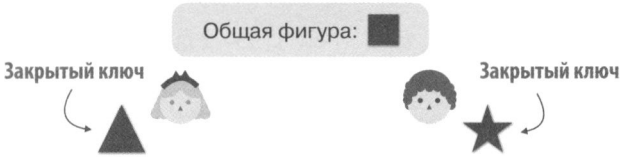


Рис. 1.5. Первый этап обмена ключами Диффи — Хеллмана: две стороны создают закрытые ключи. Королева Алиса выбирает в качестве закрытого ключа треугольник, а лорд Боб — звезду



Рис. 1.6. Второй этап обмена ключами Диффи — Хеллмана: стороны обмениваются своими открытыми ключами, полученными в результате комбинирования закрытых ключей с общей фигурой

Теперь становится понятно, почему этот алгоритм называется алгоритмом с открытым ключом: он требует *пары ключей*, состоящей из закрытого и открытого ключей. Последний шаг алгоритма обмена ключами довольно прост. Королева Алиса берет открытый ключ лорда Боба и объединяет его со своим закрытым ключом. Лорд Боб делает то же самое с открытым ключом королевы Алисы. У обеих сторон должен получиться одинаковый результат. В нашем примере это фигура, сочетающая в себе звезду, квадрат и треугольник (рис. 1.7).



Рис. 1.7. Заключительный этап обмена ключами, на котором обе стороны составляют один и тот же общий секрет. Его нельзя получить, когда видны только открытые ключи

Теперь участникам протокола предстоит использовать общий секрет. Несколько примеров этого вы увидите далее в книге. Тем не менее самый очевидный сценарий — задействовать общий секрет в алгоритме, требующем его применения. Например, королева Алиса и лорд Боб могут теперь использовать общий секрет

в качестве ключа для шифрования сообщений посредством примитива симметричного шифрования. Напомню следующее.

1. Алиса и Боб обмениваются своими открытыми ключами, маскирующими их закрытые ключи.
2. Имея открытый ключ другой стороны и свой личный ключ, они могут вычислить общий секрет.
3. У противника, наблюдающего за обменом открытыми ключами, недостаточно информации для вычисления общего секрета.

ПРИМЕЧАНИЕ

В этом примере последний пункт легко обойти. Даже не зная никаких закрытых ключей, мы можем объединить открытые ключи и получить общий секрет. К счастью, это всего лишь условность нашей аналогии, которая довольно хорошо показывает принцип работы обмена ключами.

На практике обмен ключами довольно небезопасен. Немного поразмыслив, вы поймете почему.

Так как королева Алиса принимает любой отправленный ей открытый ключ за открытый ключ лорда Боба, я могу перехватить обмен и заменить этот ключ своим, что позволит мне выдать себя за лорда Боба (и то же самое можно сделать в отношении лорда Боба). Так называемый *посредник* (man-in-the-middle, MITM) может успешно атаковать данный протокол. Как это исправить? В последующих главах мы увидим, что нужно либо дополнить этот протокол другим криптографическим примитивом, либо заранее знать, каков открытый ключ лорда Боба. Но разве тогда не получается, что мы вернулись к самому началу?

Раньше королеве Алисе и лорду Бобу нужно было знать общий секрет. Теперь им требуется знать открытые ключи друг друга. Но как? Неужели это снова проблема курицы и яйца? Ну, вроде того. Как мы увидим далее, на практике криптография с открытым ключом не решает проблемы доверия, но упрощает его установление, особенно при большом количестве сторон.

Остановимся на этом и перейдем к следующему разделу, поскольку в главе 5 мы подробнее разберем обмен ключами. Нам осталось раскрыть еще несколько примитивов асимметричного шифрования (рис. 1.8), чтобы завершить экскурс в криптографию реального мира.

1.4.2. Асимметричное шифрование, не похожее на симметричное

За изобретением алгоритма обмена ключами быстро последовало изобретение *алгоритма RSA*, названного в честь Рона Ривеста, Ади Шамира и Леонарда Адлемана. RSA состоит из двух примитивов: алгоритма шифрования с открытым ключом (асимметричное шифрование) и электронной (цифровой) подписи. Оба примитива относятся к большому классу криптографических алгоритмов — *асимметричной криптографии*. В этом разделе разберем, как работают эти примитивы и в чем их польза.

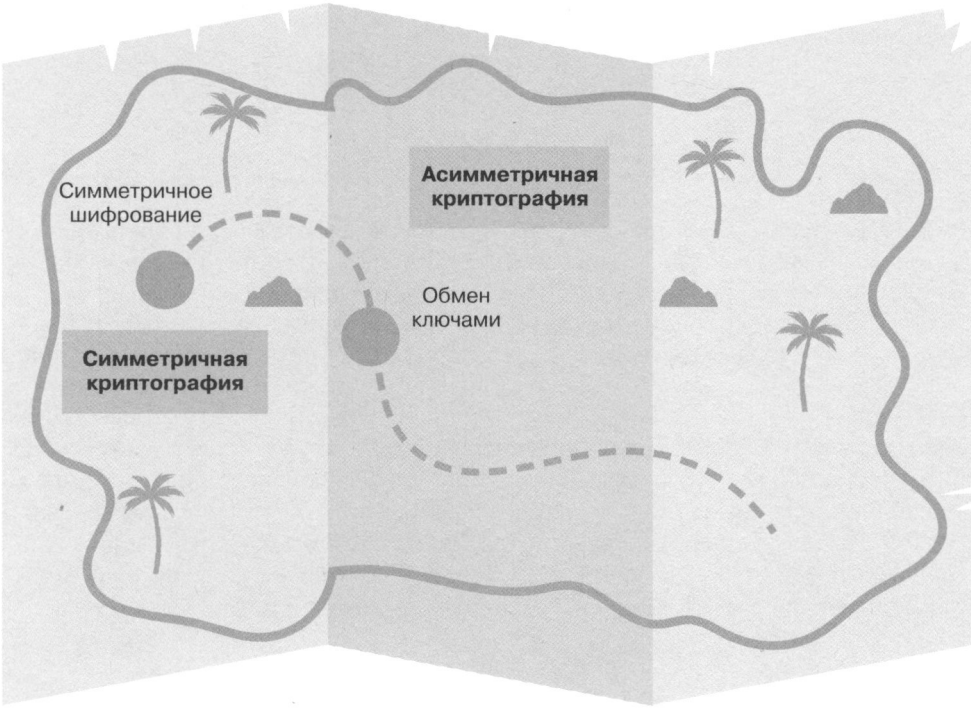


Рис. 1.8. Криптографические алгоритмы, о которых мы уже знаем. Два больших класса криптографических алгоритмов — симметричная криптография (с симметричным шифрованием) и асимметричная криптография (с обменом ключами)

Первый примитив, асимметричное шифрование, имеет ту же задачу, что и алгоритм симметричного шифрования, о котором говорилось ранее: он позволяет шифровать сообщения для обеспечения конфиденциальности. Однако, в отличие от симметричного шифрования, в котором два участника шифруют и расшифровывают сообщения с помощью одного и того же симметричного ключа, асимметричное шифрование работает совсем по-другому.

- В нем используются два разных ключа — открытый и закрытый.
- Оно предоставляет асимметричный доступ к данным: любой может шифровать с помощью открытого ключа, но расшифровывать сообщения может только владелец закрытого ключа.

Воспользуемся простой аналогией, чтобы понять, как можно применить асимметричное шифрование. Начнем с нашей подруги королевы Алисы, у которой есть закрытый ключ и соответствующий открытый ключ. Представим ее открытый ключ в виде распахнутого сундука, рыться в котором она разрешает всем желающим (рис. 1.9).



Рис. 1.9. Для того чтобы использовать асимметричное шифрование, королева Алиса должна сначала предоставить свой открытый ключ (здесь — распахнутый сундук). Теперь любой может применить открытый ключ для шифрования адресованных ей сообщений. А у нее должна быть возможность расшифровать их с помощью соответствующего закрытого ключа

Теперь и вы, и я, и все желающие могут зашифровать сообщение для королевы Алисы, используя ее открытый ключ. Представьте, что вы кладете свое сообщение в открытый сундук, а затем закрываете его. Теперь никто, кроме королевы Алисы, не сможет открыть этот сундук. Он эффективно обеспечивает секретность сообщения для сторонних наблюдателей. Закрытый сундук (или зашифрованное содержимое) можно отправить королеве Алисе, которая, в свою очередь, может применить свой закрытый ключ (известный только ей, помните) для расшифровки (рис. 1.10).

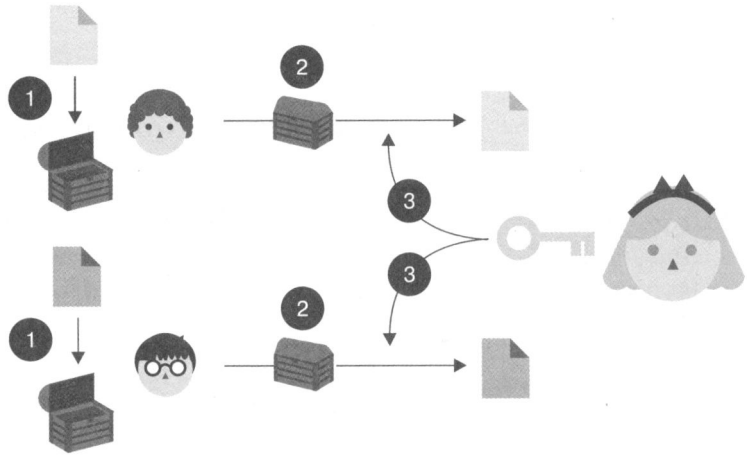


Рис. 1.10. Асимметричное шифрование: любой человек может воспользоваться открытым ключом королевы Алисы для шифрования отправляемых ей сообщений *(1)*. Получив эти сообщения *(2)*, она может расшифровать их содержимое посредством своего закрытого ключа *(3)*. В процессе отправки сообщений королеве Алисе ни у кого нет возможности увидеть их содержание

Добавим на нашу карту (рис. 1.11) рассмотренный сейчас криптографический примитив. Осталась еще одна точка для завершения экскурса в криптографию реального мира!

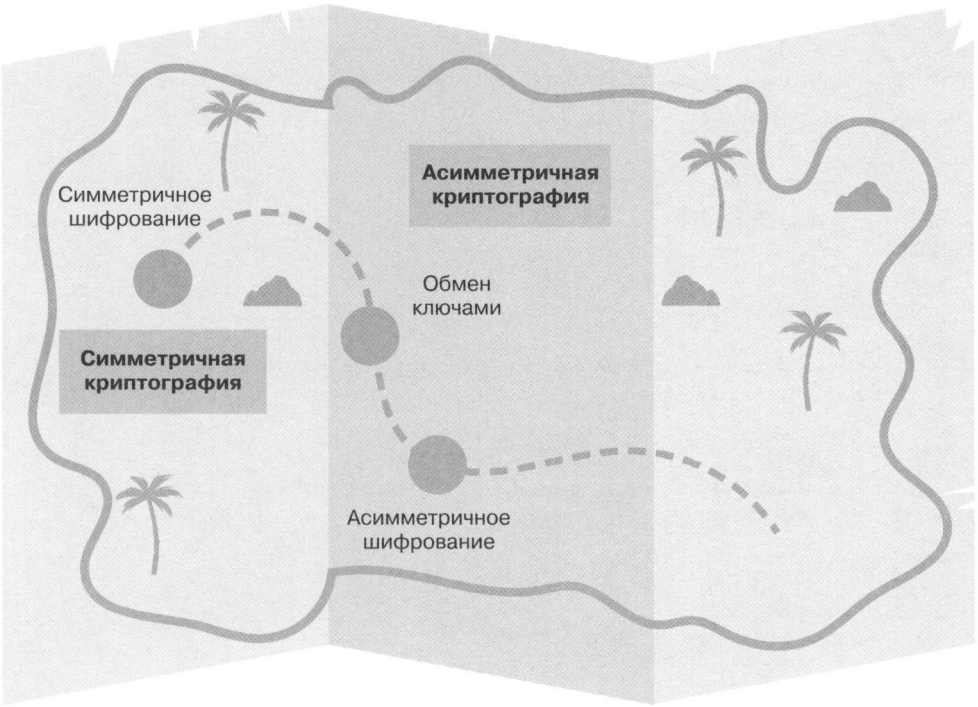


Рис. 1.11. Криптографические алгоритмы, о которых мы уже знаем: два больших класса криптографических алгоритмов — симметричная криптография (с симметричным шифрованием) и асимметричная криптография (с обменом ключами и асимметричным шифрованием)

1.4.3. Цифровые подписи, прямо как подписи на бумаге

Как я упомянул ранее, RSA — это алгоритм не только асимметричного шифрования, но и *цифровой подписи*. Изобретение такого криптографического примитива, как цифровая подпись, очень сильно посодействовало установлению доверия между Алисами и Бобами нашего мира. Цифровая подпись похожа на настоящую, например на ту, которую ставят в договоре на аренду квартиры.

«А что, если кто-то подделает мою подпись?» — спросите вы, и действительно, в реальном мире подписи на бумаге не особо-то и защищают. В то же время криптографические подписи, выполняющие ту же функцию, что и реальные, предоставляют цифровой сертификат с вашим именем. Таким образом, криптографическую подпись *нельзя подделать* и ее легко проверить, что довольно полезно по сравнению с устаревшими подписями, которые раньше ставили на чеках!

На рис. 1.12 представлен протокол, в котором королева Алиса хочет показать лорду Дэвиду, что доверяет лорду Бобу. Это типичный пример того, как можно установить доверие в многосторонней среде и чем при этом может помочь асимметричная криптография. Подписав лист бумаги со словами: «Я, королева Алиса,

доверяю лорду Бобу», королева Алиса может ясно дать понять лорду Дэвиду, что лорду Бобу можно доверять. Если лорд Дэвид уже доверяет королеве Алисе и ее алгоритму подписи, то он может начать доверять и лорду Бобу.



Рис. 1.12. Лорд Дэвид уже доверяет королеве Алисе. Безопасно ли для него доверять лорду Бобу, которому доверяет королева Алиса?

Если говорить точнее, королева Алиса может использовать схему подписи RSA и свой закрытый ключ, чтобы подписать сообщение: «Я, королева Алиса, доверяю лорду Бобу». При этом сгенерируется подпись в виде случайного шума (рис. 1.13).

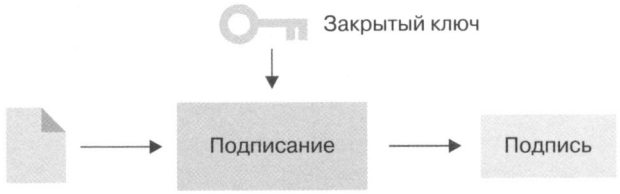


Рис. 1.13. Для подписания сообщения королеве Алисе нужно применить свой закрытый ключ и сгенерировать подпись

Любой человек может *проверить подпись*, объединив:

- открытый ключ Алисы;
- подписанное сообщение;
- подпись.

Результатом будет либо «истина» (подпись действительна), либо «ложь» (подпись недействительна), как показано на рис. 1.14.



Рис. 1.14. Для проверки подписи, полученной от королевы Алисы, необходимы ее открытый ключ и подписанное сообщение. В результате подпись признается либо действительной, либо недействительной

Мы рассмотрели три асимметричных примитива:

- обмен ключами Диффи — Хеллмана;
- асимметричное шифрование;
- цифровые подписи RSA.

Это наиболее известные и часто используемые примитивы в асимметричной криптографии. Возможно, не совсем очевидна их полезность при решении реальных проблем. Однако заверяю вас: они применяются каждый день во многих приложениях для защиты от широкого спектра угроз. Пришло время закончить нашу карту только что рассмотренным криптографическим алгоритмом (рис. 1.15).

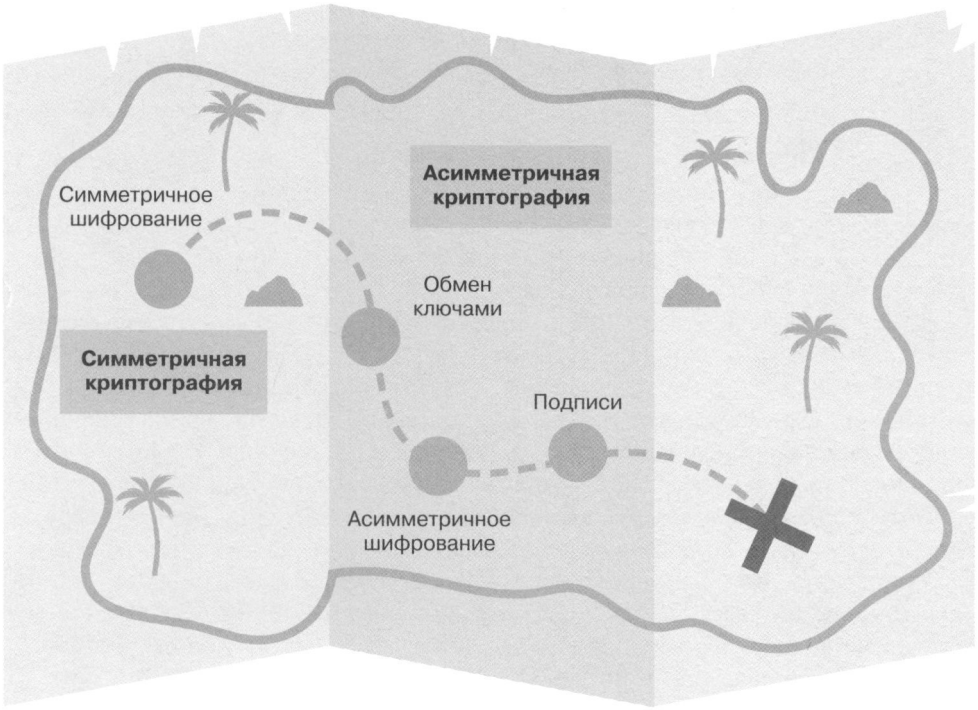


Рис. 1.15. Все рассмотренные симметричные и асимметричные алгоритмы

1.5. КЛАССИФИКАЦИЯ И ОБОБЩЕНИЕ В ОБЛАСТИ КРИПТОГРАФИИ

В предыдущем разделе мы рассмотрели два больших класса алгоритмов:

- *симметричную криптографию* (или *криптографию с секретным ключом*), в которой используется один секрет. Если он известен нескольким участникам, то называется *общим*;

- *асимметричную криптографию* (или *криптографию с открытым ключом*), в которой стороны имеют асимметричный доступ к секретам. Например, одним известен только открытый ключ, а другим — и открытый, и закрытый.

Криптографические примитивы не делятся лишь на симметричную и асимметричную криптографию, и в то же время довольно сложно выделять еще какие-либо подобласти. Однако, как вы увидите далее, бóльшая часть нашей книги посвящена симметричным и асимметричным примитивам и их применению, ведь многое из того, что на практике используется в криптографии в настоящее время, относится именно к этим подобластям. Тем не менее криптографию можно разделить также:

- на *математические конструкции*, основанные на таких математических задачах, как факторизация чисел (пример такого типа конструкций — алгоритм RSA для цифровых подписей и асимметричного шифрования);
- *эвристические конструкции*, основанные на наблюдениях и статистическом анализе, выполняемом криптоаналитиками (пример — AES для симметричного шифрования).

Одним из оснований такого разделения на категории является скорость, поскольку конструкции, основанные на математике, часто намного медленнее в разработке, чем конструкции, основанные на эвристике. Если конкретнее: симметричные конструкции чаще всего основаны на эвристике (то, что кажется работающим), в то время как большинство асимметричных конструкций базируются на математических задачах (то, что считается сложным).

Нам трудно строго распределить все, что может предложить криптография, по категориям. В каждой книге или в курсе по этому предмету даются различные определения и классификации. В конце концов, эти различия не слишком полезны для нас, поскольку мы будем рассматривать большинство криптографических примитивов как уникальные инструменты, которые дают уникальные обещания относительно *надежности защиты*. В свою очередь, можно использовать многие из этих инструментов в качестве строительных блоков для создания протоколов. Таким образом, чтобы понять, как примитивы защищают протоколы, с которыми мы взаимодействуем, очень важно разобраться, как работает каждый из них и насколько надежную защиту они обеспечивают. По этой причине в части I будут рассмотрены наиболее полезные криптографические примитивы и их свойства безопасности.

Многие концепции, изложенные в книге, на первый взгляд могут показаться довольно сложными. Но, как и при обучении всему на свете, чем больше мы прочитаем о них и чем чаще увидим их работу в контексте, тем более естественными и легче выделяемыми из ряда других они станут. Роль этой книги заключается в том, чтобы помочь вам вывести тезисы, позволить создать мысленную модель того, как работают криптографические конструкции, и понять, как их можно объединить в надежные протоколы. Я буду часто говорить о зоне соприкосновения конструкций и приводить реальные примеры их использования и комбинирования.

Раньше определение криптографии было простым: королева Алиса и лорд Боб хотят обменяться секретными сообщениями. Теперь это не так. Сегодня криптографию довольно сложно описать. Она органично развивалась на основе

открытий, инновационных решений и практических потребностей. В конце концов, криптография — это то, что помогает усилить протокол, чтобы он мог работать в небезопасной среде.

Для того чтобы полностью понимать пользу криптографии, нам важно, какой перечень задач выполняют протоколы. Большинство криптографических примитивов и протоколов, о которых будет идти речь в книге, обладают одним из следующих свойств безопасности или обоими.

- *Конфиденциальность* — маскировка и защита информации от посторонних глаз. Например, шифрование маскирует сообщения при передаче.
- *Аутентификация* — идентификация собеседника. Например, она может быть полезна для того, чтобы убедиться, что получаемые нами сообщения на самом деле отправлены королевой Алисой.

Конечно, это все еще сильно упрощенное представление о криптографии. В большинстве случаев детали заключаются в том, как гарантируется надежность примитивов. В зависимости от того, каким образом мы используем криптографический примитив в протоколе, он будет обеспечивать различные свойства безопасности.

В разных главах книги мы будем изучать новые криптографические примитивы и то, как их можно комбинировать между собой для обеспечения таких свойств безопасности, как конфиденциальность и аутентификация. Пока же необходимо в должной мере осознать тот факт, что криптография занимается обеспечением протокола средствами защиты в небезопасной среде. Хотя нельзя четко определить, кем являются наши противники, мы можем предположить, что это те, кто пытается взломать наш протокол: другая сторона, наблюдатель, посредник, что в достаточной мере отражает то, кем они могут быть в реальной жизни. В конечном счете криптография — это практическая область, созданная для защиты от злоумышленников, состоящих из плоти, костей и битов.

1.6. КРИПТОГРАФИЯ В ТЕОРИИ И КРИПТОГРАФИЯ РЕАЛЬНОГО МИРА

В 1993 году Брюс Шнайер выпустил книгу «Прикладная криптография» для разработчиков и инженеров, работающих над приложениями с криптографией. Где-то с 2012 года Кенни Патерсон и Найджел Смарт начали проводить ежегодные конференции Real World Crypto, ориентированные на ту же аудиторию. Но что имеется в виду под прикладной криптографией и криптографией реального мира? Существует более одного типа криптографии?

Прежде чем ответить на эти вопросы, необходимо определить, что такое *теоретическая криптография*. Над теоретической криптографией работают криптографы и криптоаналитики, основную часть которых составляют представители академических кругов из университетов. Встречаются среди них также практикующие специалисты или люди из определенных департаментов правительства.

Они исследуют все и вся в области криптографии. Результаты их работы распространяются на международном уровне посредством публикаций в журналах и презентаций на конференциях. Однако не все, что делается в рамках теоретической криптографии, можно без сомнений признать полезным или практичным. Часто не публикуются доказательство концепции или код, хотя это все равно становится неважным, если ни один компьютер не обладает достаточной мощностью для их практического применения. Тем не менее некоторые достижения теоретической криптографии оказываются настолько полезными и практичными, что переходят на другую сторону.

Другой стороной является мир *прикладной криптографии*, или *криптографии реального мира*. Это основа защиты всех наших приложений. Несмотря на то что часто кажется, будто ее нет совсем или она практически прозрачна, криптография присутствует, когда вы входите в свой интернет-банкинг. Она с вами, когда вы переписываетесь с друзьями. Она помогает защититься в случае потери телефона. Она используется повсеместно, потому что, к сожалению, злоумышленники находятся повсюду и активно пытаются изучить наши системы и нанести им вред. В данной отрасли обычно работают практикующие специалисты, которые время от времени проверяют алгоритмы и разрабатывают протоколы с помощью академического сообщества. Результаты их работы часто распространяются на конференциях, в блогах и через программное обеспечение с открытым исходным кодом.

Как правило, практическую криптографию очень волнуют аспекты реального мира: какой точный уровень защиты обеспечивается тем или иным алгоритмом? Сколько времени требуется для выполнения алгоритма? Каков размер входных и выходных данных, требуемых примитивом? Как вы уже могли догадаться, криптография реального мира — это главная тема данной книги. Теоретическая криптография является основой других книг, мы же взглянем на то, что в этой области происходит, в последних главах. Будьте готовы удивляться, потому что своими глазами увидите реальную криптографию завтрашнего дня.

Возможно, теперь вы задаетесь вопросом: как разработчики и инженеры решают, какую криптографию использовать для своих реальных приложений?

1.7. ОТ ТЕОРИИ К ПРАКТИКЕ: ВЫБЕРИ СЕБЕ ПРИКЛЮЧЕНИЕ

> На вершине сидят криптоаналитики, которые предлагают и решают сложные математические задачи... а в самом низу — инженеры-программисты, которые хотят зашифровать некоторые данные.
>
> *Тай Дуонг, So you want to roll your own crypto? (2020)*

За все годы обучения и работы в сфере криптографии я так и не увидел единой модели использования криптографических примитивов в реальных приложениях. Все происходит довольно хаотично. Перед тем как примитив, разработанный в теории, будет принят реальным миром, множество людей должны его обработать

и превратить в нечто пригодное к применению, а иногда и более безопасное для общественности в целом. Как можно объяснить вам это?

Слышали ли вы о книгах «Выбери себе приключение»? Это старая серия книг, в которой читатель сам выбирает сюжет. Принцип прост: человек читает первый раздел книги, а в конце его может выбрать дальнейший путь из предложенных вариантов. Каждый вариант соотносится с определенным номером раздела, к которому можно перейти согласно сделанному выбору. Итак, я проделаю то же самое здесь! Читайте текст далее и следуйте указаниям.

С чего все начинается. Кто вы? Криптограф Алиса? Работающий в частном бизнесе Дэвид, которому необходимо найти решение проблемы? Или Ева из одной правительственной структуры, все внимание которой занято криптографией?

- Вы Алиса — перейдите к шагу 1.
- Вы Дэвид — перейдите к шагу 2.
- Вы Ева — перейдите к шагу 3.

Шаг 1. Исследователи должны проводить исследования. Вы являетесь научным сотрудником университета, или исследовательской группы частной компании (либо некоммерческой организации), или государственной исследовательской организации, например NIST или Агентства по национальной безопасности (АНБ). Следовательно, финансирование может поступать из разных мест и тем самым поощрять вас к проведению различных исследований.

- Вы изобретаете новый примитив — перейдите к шагу 4.
- Вы изобретаете новую конструкцию — перейдите к шагу 5.
- Вы запускаете открытый конкурс — перейдите к шагу 6.

Шаг 2. У бизнеса есть потребность. На работе что-то случается, и возникает потребность в новом стандарте. Например, Альянс Wi-Fi (Wi-Fi Alliance) — это финансируемая заинтересованными компаниями некоммерческая организация, которая занимается созданием стандартов, касающихся протокола Wi-Fi. Другой пример — банки, объединившиеся для разработки стандарта безопасности индустрии платежных карт (Payment Card Industry Data Security Standard, PCI DSS). Данный стандарт вводит алгоритмы и протоколы, которые необходимо использовать при работе с номерами кредитных карт.

- Вы решили профинансировать некоторые очень нужные исследования — перейдите к шагу 1.
- Вы решили стандартизировать новый примитив или протокол — перейдите к шагу 5.
- Вы запускаете открытый конкурс — перейдите к шагу 6.

Шаг 3. У правительства есть потребность. Вы работаете на правительство своей страны, и вам нужно выпустить новую криптографию. К примеру, перед NIST поставили задачу опубликовать *федеральные стандарты обработки информации* (Federal

Information Processing Standards, FIPS), которые устанавливают, какие криптографические алгоритмы могут использоваться компаниями, работающими с правительством США. Несмотря на то что многие из них успешны, а люди склонны с большим доверием относиться к продвигаемым правительственными агентствами стандартам, о неудачах (к сожалению) тоже можно многое рассказать.

В 2013 году, после разоблачений, которые сделал Эдвард Сноуден, выяснилось, что АНБ целенаправленно и успешно добилось включения в стандарты бэкдора (см. *Dual EC: A Standardized Back Door* Бернштейна и др.), который содержал скрытый параметр, позволяющий АНБ, и только АНБ, отслеживать пользователей. *Бэкдоры* можно рассматривать как волшебные пароли, позволяющие правительству (и только ему, предположительно) взламывать алгоритмы шифрования. С тех пор криптографическое сообщество практически перестало доверять правительственным стандартам и предложениям. Недавно, в 2019 году, выяснилось, что российский стандарт ГОСТ имел такие же проблемы.

> Криптографы давно подозревали, что агентство заложило уязвимости в стандарт, который был принят в 2006 году Национальным институтом стандартов и технологий, а затем Международной организацией по стандартизации, членами которой являются 163 страны. Засекреченные служебные записки АНБ подтверждают, что критический дефект, обнаруженный двумя криптографами Microsoft в 2007 году, был разработан самим агентством. АНБ написало стандарт и агрессивно продвигало его на международном уровне, негласно называя эти усилия вызовом изяществу.
>
> *«Нью-Йорк таймс» (N. S. A. Able to Foil Basic Safeguards of Privacy on Web, 2013)*

- Вы финансируете исследование — перейдите к шагу 1.
- Вы организуете открытый конкурс — перейдите к шагу 6.
- Вы добиваетесь стандартизации используемого вами примитива или протокола — перейдите к шагу 7.

Шаг 4. Предлагается новая концепция. Как исследователю, вам удается сделать невозможное: вы придумываете новую концепцию. Конечно, кто-то уже задумывался о шифровании, но в криптографии каждый год предлагаются новые примитивы. Некоторые из них окажутся нереализуемыми, а часть в конечном итоге будет разгадана. Возможно, в вашем предложении есть реальная конструкция, а может быть, вам придется подождать и посмотреть, сможет ли кто-нибудь предложить что-то работающее.

- Ваш примитив реализуется — перейдите к шагу 5.
- Ваш примитив оказывается нереализуемым — вернитесь к началу.

Шаг 5. Предлагается новая конструкция или протокол. Криптограф или группа криптографов предлагает новый алгоритм, который реализует концепцию. Например, AES является конкретизацией схемы шифрования. (AES был предложен Винсентом Риджменом и Джоан Демен, которые назвали конструкцию сокращением своих фамилий — Rijndael.) Что дальше?

- Кто-то берет за основу вашу конструкцию — перейдите к шагу 5.
- Вы участвуете в открытом конкурсе и побеждаете! Перейдите к шагу 6.
- Ваша работа вызвала большой шум и становится стандартом! Перейдите к шагу 7.
- Вы решаете запатентовать свою конструкцию — перейдите к шагу 8.
- Вы или кто-то другой решаете, что будет интересно реализовать конструкцию, — перейдите к шагу 9.

Шаг 6. Алгоритм побеждает в конкурсе. Процесс, который криптографы любят больше всего, — это открытый конкурс! Например, AES был конкурсом, в котором участвовали исследователи со всего мира. В результате многоступенчатого анализа десятков поданных заявок и с помощью криптоаналитиков (что может занять годы) список был сокращен до нескольких кандидатов (в случае с AES — до одного), которые затем вошли в стандарт.

- Вам повезло, в многолетнем конкурсе ваша конструкция победила! Перейдите к шагу 7.
- К сожалению, вы проиграли — вернитесь к началу.

Шаг 7. Алгоритм или протокол стандартизируется. Стандарт обычно публикуется правительством или организацией по стандартизации. Цель публикации состоит в том, чтобы избежать недопонимания и разногласий и тем самым максимизировать функциональную совместимость. Так, NIST регулярно публикует криптографические стандарты. Известной организацией по стандартизации в области криптографии является также Инженерный совет Интернета (Internet Engineering Task Force, IETF), который стоит за выпуском многих стандартов в Интернете (TCP, UDP, TLS и др.) и о котором вы много узнаете в этой книге. В IETF стандарты называются *рабочими предложениями* (Request for Comments, RFC) и могут быть разработаны практически любым желающим.

> Для того чтобы подкрепить идею отсутствия голосования, мы также переняли традицию «гула». Когда, например, мы проводим очные совещания и председатель рабочей группы хочет «понять отношение присутствующих» к какому-либо вопросу, он может попросить каждую сторону выразить его (либо «за», либо «против») гулом, а не поднятием рук.
>
> *RFC 7282 («О консенсусе и гуле в IETF», 2014)*

Иногда компания публикует стандарт напрямую. Так, RSA Security LLC, финансируемая создателями алгоритма RSA, выпустила серию из 15 документов под названием «Стандарты криптографии с открытым ключом» (Public Key Cryptography Standards, PKCS), в которых легитимизовала алгоритмы и методы, используемые компанией в то время. Сейчас такое встречается довольно редко, и многие компании стандартизируют свои протоколы или алгоритмы через IETF в виде RFC, а не внутреннего документа.

- Ваш алгоритм или протокол реализован — перейдите к шагу 9.
- Никого не волнует ваш стандарт — вернитесь к началу.

Шаг 8. Истечение срока действия патента. В криптографии патент обычно означает, что алгоритм никто не будет использовать. Возобновление интереса к примитиву нередко наблюдается после истечения срока действия патента на него. Самым популярным примером, вероятно, являются подписи Шнорра. Данный алгоритм был первым претендентом на то, чтобы стать самой популярной схемой подписи, пока Шнорр не запатентовал его в 1989 году. Это привело к тому, что NIST стандартизировал менее безопасный алгоритм цифровой подписи (Digital Signature Algorithm, DSA), который стал тогда главной схемой подписи, однако сейчас не находит широкого применения. Срок действия патента на подписи Шнорра истек в 2008 году, и с тех пор алгоритм вновь начал обретать популярность.

- Прошло слишком много времени, и о вашем алгоритме забыли навсегда — вернитесь к началу.
- Ваша конструкция стала идейной основной для множества других конструкций — перейдите к шагу 5.
- Теперь люди хотят использовать вашу конструкцию, но не раньше ее официальной стандартизации — перейдите к шагу 7.
- Некоторые разработчики реализуют ваш алгоритм! Перейдите к шагу 9.

Шаг 9. Реализация конструкции или протокола. Перед теми, кто хочет реализовать алгоритм, стоит сложная задача: им необходимо не только проанализировать документ или стандарт (хотя и *предполагается*, что стандарты предназначены для них), но и сделать его простым и безопасным для применения. Не всегда это просто, поскольку в том, как используется криптография, может возникнуть множество критических ошибок.

- Кто-то решает, что отсутствие у алгоритма статуса стандарта усложняет его реализацию и пришло время исправить это. Перейдите к шагу 7.
- На вашу криптографическую библиотеку обрушилось всеобщее внимание! Перейдите к шагу 10.

Шаг 10. Разработчик использует протокол или примитив в приложении. Перед разработчиком стоит определенная задача и, похоже, ваша криптографическая библиотека решает ее проще простого!

- Примитив устраняет потребность, но у него нет статуса стандарта. Это не очень хорошо. Перейдите к шагу 7.
- Хотел бы я, чтобы это было написано на моем языке программирования. Перейдите к шагу 9.
- Я неправильно использовал библиотеку, или конструкция взломана. Игра окончена.

Получилось! Существует множество способов перехода примитива в реальный мир. Лучший вариант — многолетний анализ, признание его просто реализуемым стандартом и наличие качественных библиотек. Худший вариант — плохой алгоритм с плохой реализацией. На рис. 1.16 проиллюстрирован предпочтительный вариант.

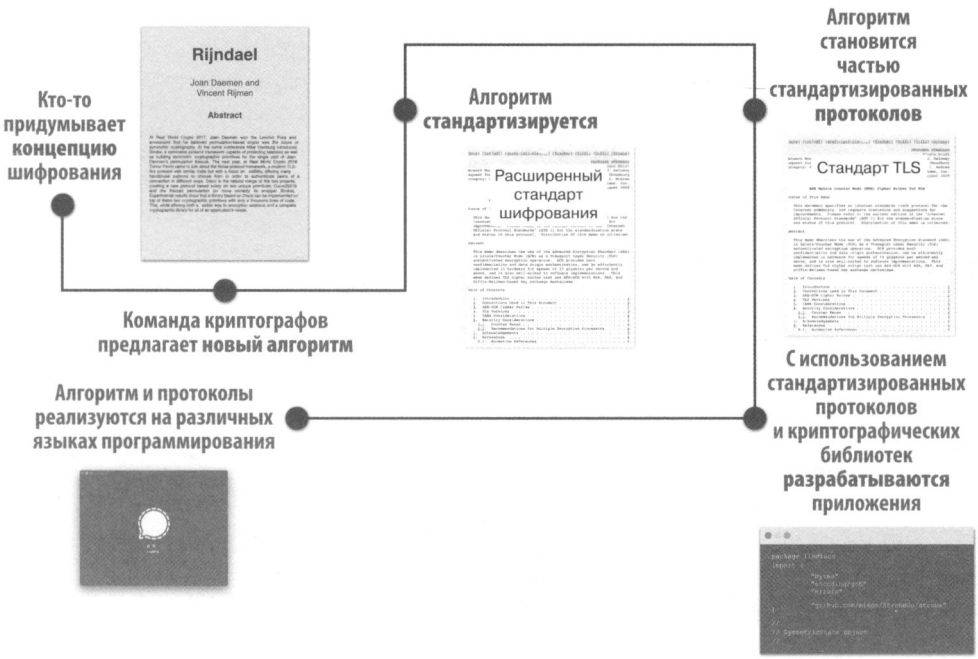


Рис. 1.16. Идеальный жизненный цикл криптографического алгоритма начинается с технического описания концепции, конкретизированной криптографами. Например, AES — это конкретизация концепции симметричного шифрования (существует множество других алгоритмов симметричного шифрования). Затем конструкция может быть стандартизирована: все соглашаются реализовывать ее определенным образом, чтобы обеспечить максимальную функциональную совместимость. В конце концов создается поддержка алгоритма путем реализации стандарта на различных языках

1.8. ПРЕДУПРЕЖДЕНИЕ

> Любой человек, от самого невежественного любителя до лучшего криптографа, может создать алгоритм, который сам не сможет взломать.
>
> *Брюс Шнайер (Memo to the Amateur Cipher Designer, 1998)*

Должен предупредить вас, что искусство криптографии трудно освоить. Было бы неразумно полагать, что вы сможете создавать сложные криптографические протоколы после того, как прочтете эту книгу. Цель данного путешествия — передать вам знания, показать ваши возможности и продемонстрировать, как все работает. Однако оно не сделает вас великим криптографом.

Эта книга — не святой Грааль. Ее страницы научат вас самому важному: нельзя отправляться в настоящее приключение в одиночку. Драконы убивают, и, чтобы победить их, не обойтись без поддержки. Другими словами, криптография сложна и одна лишь эта книга не даст вам всех навыков, необходимых для полноценной

работы с ней. Для создания комплексных систем требуются специалисты, которые годами изучают свое дело. Тем не менее вы научитесь распознавать ситуации, когда следует использовать криптографию, а в подозрительных случаях — какие криптографические примитивы и протоколы доступны для решения возникающих проблем и как все эти криптографические алгоритмы работают на самом деле. Теперь, когда вы предупреждены, можете переходить к следующей главе.

РЕЗЮМЕ

- Протокол — это подробная инструкция, с помощью которой несколько сторон пытаются достичь какой-либо цели, например обменяться конфиденциальными сообщениями.
- Криптография — это усиление протоколов для обеспечения их безопасности в небезопасной среде. Для этого часто требуются секреты.
- Криптографический примитив — это тип криптографического алгоритма. Например, симметричное шифрование — это криптографический примитив, а AES — конкретный алгоритм симметричного шифрования.
- Одним из способов классификации криптографических примитивов является разделение их на два типа: симметричную и асимметричную криптографию. Симметричная криптография использует один ключ (что можно было увидеть на примере симметричного шифрования), в то время как асимметричная криптография работает с различными ключами (что можно было увидеть на примерах обмена ключами, асимметричного шифрования и простановки цифровых подписей).
- Криптографические свойства трудно классифицировать, но они часто направлены на обеспечение одной из двух характеристик: аутентификации или конфиденциальности. Аутентификация — это проверка подлинности чего-либо или кого-либо, а конфиденциальность — обеспечение тайны данных или личности.
- Важность криптографии реального мира обусловлена ее повсеместным использованием в прикладных технологиях, в то время как теоретическая криптография зачастую менее полезна на практике.
- Большинство криптографических примитивов, приведенных в этой книге, были согласованы в результате длительных процессов стандартизации.
- Криптография сложна, при реализации или использовании криптографических примитивов возникает множество опасностей.

2 Хеш-функции

https://t.me/it_boooks/2

В этой главе

- ✓ Хеш-функции и их свойства безопасности.
- ✓ Самые часто используемые хеш-функции.
- ✓ Другие виды хеширования.

Присвоение глобальных уникальных идентификаторов чему бы то ни было — такова обязанность первой криптографической конструкции, которую мы рассмотрим в этой главе, — *хеш-функции*. В криптографии хеш-функции используются повсеместно, во всех сферах! Если говорить простым языком о принципе их работы, они принимают на вход любые данные и выдают в ответ уникальную строку байтов. При одинаковых входных данных хеш-функция всегда воспроизводит одну и ту же строку байтов. Незначительная на первый взгляд, эта простая деталь чрезвычайно полезна для построения многих других криптографических конструкций. В этой главе вы узнаете все, что нужно знать о хеш-функциях и причине их универсальности.

2.1. ЧТО ТАКОЕ ХЕШ-ФУНКЦИЯ

Перед вами кнопка загрузки, занимающая значительную часть страницы. На ней большими буквами написано: DOWNLOAD (СКАЧАТЬ), и похоже, что если вы на нее нажмете, то она перенаправит вас на другой сайт, содержащий необходимый файл. Под кнопкой располагается длинная непонятная строка из букв:

```
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

Перед ней стоит что-то похожее на аббревиатуру: `sha256sum`. Звучит знакомо? Возможно, раньше вы скачивали что-то и видели такую странную строку (рис. 2.1).



Рис. 2.1. Веб-страница, ссылающаяся на внешний веб-сайт, содержащий файл. Внешний сайт не может изменить содержимое скачиваемого файла, поскольку исходная страница предоставляет хеш, или дайджест, файла и тем самым обеспечивает его целостность

Вы когда-нибудь задумывались над тем, что можно сделать с этой длинной строкой?

1. Нажмите кнопку, чтобы скачать файл.
2. Используйте алгоритм SHA-256 для *хеширования* скачиваемого файла.
3. Сравните результат на выходе (дайджест) с длинной строкой, отображаемой на веб-странице.

Это позволит проверить, правильный ли файл вы скачали.

ПРИМЕЧАНИЕ

Результат на выходе, или выход, хеш-функции часто называют *дайджестом* или *хешем*. В этой книге я употребляю эти два слова как синонимы. В других источниках выход может называться *контрольной суммой* или просто *суммой*, однако я избегаю этих терминов, поскольку они используются прежде всего для описания некриптографических хеш-функций, что может привести к еще большей путанице. Просто имейте в виду, что в разных кодовых базах или документах применяются разные термины.

Захешировать что-нибудь можно с помощью популярной библиотеки OpenSSL. Она предлагает многофункциональный интерфейс командной строки (Command-Line Interface, CLI), который по умолчанию присутствует в ряде систем, включая macOS. Например, можно открыть терминал и написать следующую строку:

```
$ openssl dgst -sha256 скачиваемый_файл
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

В этой команде мы применили хеш-функцию SHA-256 для преобразования входных данных (скачиваемого файла) в уникальный идентификатор (значение, поддерживаемое командой). В чем польза этого действия? Мы обеспечиваем *целостность* и *подлинность* информации и, таким образом, можем быть уверены, что скачиваемый файл действительно является тем, что мы ожидаем получить.

Все это работает благодаря такому свойству безопасности хеш-функции, как *устойчивость к атаке нахождения второго прообраза*. Данный математический термин предполагает, что из длинного выхода хеш-функции `f63e...` невозможно найти другой файл, который после хеширования будет иметь тот же выход `f63e...`. На практике это означает, что дайджест тесно связан со скачиваемым файлом и никакой злоумышленник не сможет ввести вас в заблуждение, предоставив вам другой файл.

ШЕСТНАДЦАТЕРИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ

Длинная выходная строка `f63e...` представляет собой двоичные данные, отображаемые в *шестнадцатеричном формате* (кодировка по основанию 16, использующая числа от 0 до 9 и буквы от *a* до *f* для представления нескольких битов данных). Можно было бы отобразить двоичные данные с помощью 0 и 1 (по основанию 2), но тогда они заняли бы больше места. Шестнадцатеричная кодировка же позволяет записывать два буквенно-цифровых символа на каждые 8 бит (1 байт). Такая запись занимает меньше места и частично может быть прочитана человеком. Существуют и другие способы кодирования двоичных данных, но наиболее часто используются две кодировки: шестнадцатеричная система счисления и base64. Чем больше основание, тем меньше места требуется для отображения двоичной строки. Однако рано или поздно мы исчерпаем количество символов, читаемых человеком.

Обратите внимание на то, что этот длинный дайджест находится под управлением владельца (или владельцев) веб-страницы и может быть легко заменен любым человеком, имеющим возможность изменить веб-страницу. Таким образом, мы должны доверять странице, которая предоставила нам дайджест, ее владельцам и механизму, используемому для получения страницы, и в то же время не должны доверять странице, предоставившей файл, который мы скачали. В этом смысле *сама по себе хеш-функция не обеспечивает целостности*. Целостность и подлинность скачиваемого файла обеспечиваются дайджестом в сочетании с доверенным механизмом, предоставляемым этим дайджестом (в данном случае HTTPS). Мы поговорим об HTTPS в главе 9, а пока представьте, что он волшебным образом позволяет вам безопасно общаться с веб-сайтом.

Вернемся к нашей хеш-функции, которую можно представить в виде черного ящика (рис. 2.2). Наш черный ящик принимает на вход один набор данных и выдает один выход.



Рис. 2.2. Хеш-функция принимает на вход данные произвольной длины (файл, сообщение, видео и т. д.) и создает выход фиксированной длины (например, 256 бит для SHA-256). При хешировании одних и тех же входных данных получается одинаковый дайджест, или хеш

Входные данные этой функции могут быть любого размера. Они могут быть даже пустыми. *Выход* всегда имеет одинаковую длину и является *детерминированным*, то есть функция всегда будет давать одинаковый результат при одних и тех же входных данных. В нашем примере SHA-256 всегда дает на выходе 256 бит (32 байта), закодированных в 64 буквенно-цифровых символа в шестнадцатеричном формате. Одним из основных свойств хеш-функции является то, что алгоритм нельзя обратить, то есть нельзя обнаружить входные данные по одному лишь выходу. Таким образом, можно говорить, что хеш-функции работают в *одностороннем порядке*.

Проиллюстрировать работу хеш-функции можно на примере, в котором мы захешируем различные данные хеш-функцией SHA-256, используя командную строку и библиотеку OpenSSL. Вот что показывает терминальная сессия[1]:

```
$ echo -n "привет" | openssl dgst -sha256
e58f1e8c55fa105bdd3f40e5037eb0b039b5998d52c05e6cd98878dd2da5cab2
$ echo -n "привет" | openssl dgst -sha256
e58f1e8c55fa105bdd3f40e5037eb0b039b5998d52c05e6cd98878dd2da5cab2
$ echo -n "превет" | openssl dgst -sha256
a7e315a63a837af982cc77746b25d20137fee8554fe0406077e34a7a5f33c020
$ echo -n "это очень очень очень очень очень очень
➥ очень очень очень длинное предложение" | openssl dgst -sha256
0ccad1af727e8953050524705f9779da09c5ece5b8568a361ea78fa89dde6b24
```

Хеширование одних и тех же входных данных имеет одинаковый результат

Небольшое изменение входных данных полностью меняет выход

Размер выхода никогда не меняется и не зависит от размера данных на входе

В следующем разделе рассмотрим основные свойства безопасности хеш-функций.

2.2. СВОЙСТВА БЕЗОПАСНОСТИ ХЕШ-ФУНКЦИИ

Хеш-функции в прикладной криптографии — это конструкции, которые по определению должны обладать тремя конкретными свойствами безопасности. Как мы увидим в следующих разделах, со временем это определение изменилось. А сейчас давайте обозначим три прочные основы, из которых состоит хеш-функция.

[1] Здесь и далее, где возможно, хеши отличаются от приведенных в оригинале, так как хешировались слова на русском языке. — *Примеч. пер.*

Это важно, поскольку необходимо понимать, где хеш-функции могут быть полезны, а где они не будут работать.

Первое свойство — *устойчивость к атаке нахождения прообраза*. Оно гарантирует, что никто не сможет обратить хеш-функцию, то есть использовать выход для восстановления входных данных. На рис. 2.3 показана эта односторонность. На нем наша хеш-функция подобна блендеру: из приготовленного коктейля невозможно восстановить ингредиенты.



Рис. 2.3. Созданный хеш-функцией (представлена здесь в виде блендера) дайджест невозможно (или технически настолько сложно, что предполагается, что это никогда не произойдет) изменить и использовать для нахождения исходных входных данных

ВНИМАНИЕ

Работает ли это свойство, если входные данные небольшого размера? Допустим, это *oui* либо *non*. Кто-то легко может хешировать все возможные слова, состоящие из трех букв, и узнать, что было введено изначально. А если пространство для ввода невелико? К примеру, вы всегда хешируете варианты предложения «Я буду дома в понедельник в 3 часа ночи». В данном случае тот, кто понимает, как выглядит предложение, но не знает точно день недели или час, все равно может хешировать все возможные варианты, пока не получит правильный результат. По сути, первое свойство безопасности, связанное с первым прообразом, имеет очевидную оговорку: *невозможно скрыть что-то слишком маленькое или предсказуемое*.

Второе свойство — *устойчивость к атаке нахождения второго прообраза*. Я уже упоминал это свойство безопасности, когда говорил о защите целостности файла. Его смысл заключается в том, что, если я дам вам входные данные и их дайджест, у вас не должно быть возможности найти другие входные данные, имеющие тот же дайджест. Этот принцип наглядно иллюстрирует рис. 2.4.

Обратите внимание на то, что *мы не управляем первыми входными данными*. Это замечание важно для понимания следующего свойства безопасности хеш-функций.

Наконец, третье свойство — *устойчивость к коллизиям*. Оно гарантирует, что никто не сможет произвести два разных входных блока данных, которые после

хеширования имеют одинаковый выход (рис. 2.5). В отличие от предыдущего свойства, которое фиксирует один из входов, здесь злоумышленник может выбрать два входных блока данных.

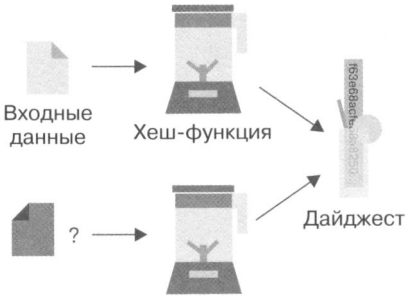


Рис. 2.4. На основании одних входных данных и их дайджеста не должно быть возможности найти другие входные данные, которые после хеширования имеют тот же выход

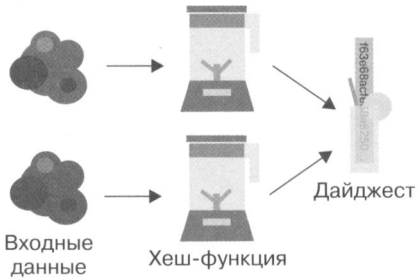


Рис. 2.5. Не должно быть ни одной возможности найти два входных блока данных (представлены *слева* в виде двух случайных сгустков данных), которые после хеширования имеют одно и то же выходное значение *(справа)*

Люди часто путают устойчивость к коллизиям и устойчивость к атаке нахождения второго прообраза. Уделите немного времени тому, чтобы понять разницу между этими двумя свойствами.

СЛУЧАЙНЫЙ ОРАКУЛ

Хеш-функции обычно разрабатываются таким образом, чтобы их дайджесты имели *непредсказуемый и случайный вид*. Это полезно, поскольку не всегда можно доказать безопасность протокола, достигаемую благодаря одному из свойств безопасности хеш-функции, о которых мы говорили, например устойчивости к коллизиям. Безопасность многих протоколов доказывается посредством *модели случайного оракула*, где используется вымышленный идеальный участник — случайный оракул. С помощью протокола такого типа можно посылать любые входные данные в качестве запросов этому случайному оракулу, который, как говорят, возвращает совершенно случайный выход и, подобно хеш-функции, дважды вернет один и тот же выход, если дважды дать ему одни и те же входные данные.

Доказательства в этой модели время от времени вызывают споры, поскольку мы не знаем наверняка, могут ли реальные хеш-функции выполнять роль таких случайных оракулов на практике. Тем не менее многие легитимные протоколы доказали свою безопасность с помощью этого метода, в результате чего хеш-функции считаются идеальнее, чем они, вероятно, есть на самом деле.

2.3. МЕТОДЫ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ДЛЯ ХЕШ-ФУНКЦИЙ

На данный момент мы рассмотрели три свойства безопасности хеш-функции:

- устойчивость к атаке нахождения прообраза;
- устойчивость к атаке нахождения второго прообраза;
- устойчивость к коллизиям.

Часто сами по себе они не имеют смысла, так как все зависит от того, как используется хеш-функция. Тем не менее, прежде чем рассматривать работу реальных хеш-функций, важно понимать некоторые ограничения перечисленных свойств безопасности.

Во-первых, эти свойства безопасности предполагают разумное применение хеш-функции. Представьте, что я хеширую либо слово «да», либо слово «нет», а затем публикую дайджест. Если вы имеете некоторое представление о том, что я делаю, то можете просто захешировать эти слова и сравнить результат с тем, что опубликовано мной. Поскольку здесь нет никаких секретов, а алгоритм используемого нами хеширования находится в свободном доступе, у вас не возникнет каких-либо проблем с тем, чтобы обнаружить исходные данные. Здесь можно подумать о слабой устойчивости хеш-функции к атаке нахождения прообраза, но я буду утверждать, что это ваши входные данные не были достаточно случайными. Более того, поскольку длина выхода хеш-функции всегда одинакова, независимо от размера входных данных, существует бесконечное количество входов, которые после хеширования имеют один и тот же выход. Опять же вы можете спросить: «Ну разве это не нарушение устойчивости к атаке нахождения второго прообраза?» Устойчивость к атаке нахождения второго прообраза просто говорит о том, что найти другой вход чрезвычайно трудно — настолько трудно, что предполагается, что это невозможно на практике, но не в теории.

Во-вторых, размер дайджеста *имеет* значение. Это отнюдь не характерная особенность хеш-функций. Во всех криптографических алгоритмах особое внимание должно уделяться размеру используемых в них параметров. Представим себе следующий пример. У нас есть хеш-функция, которая равномерно случайным образом на выходе выдает данные длиной 2 бита, то есть `00` в 25 % времени, `01` в 25 % времени и т. д. В таком случае не придется прилагать слишком много усилий, чтобы достигнуть коллизии: после хеширования нескольких случайных входных строк вы с большой вероятностью сможете найти две, которые после хеширования имеют один и тот же выход. По этой причине существует *обязательный минимальный размер выхода* хеш-функции — 256 бит, или 32 байта. При таком большом размере выхода коллизий быть не должно, если только в вычислительной технике не произойдет прорыв.

Как было выведено это число? В реальной криптографии алгоритмы ориентируются на минимальный уровень безопасности в 128 бит. Злоумышленник, который хочет взломать алгоритм, обеспечивающий 128-битную защиту, должен выполнить около 2^{128} операций (например, перебор всех возможных входных строк

длиной 128 бит потребует 2^{128} операций). Чтобы хеш-функция обеспечивала все три вышеупомянутых свойства безопасности, она должна обеспечить как минимум 128 бит защиты от трех соответствующих видов атак. Самой простой атакой обычно является поиск коллизий в связи с *ограниченностью дней рождения*.

ОГРАНИЧЕННОСТЬ ДНЕЙ РОЖДЕНИЯ

Ограниченность дней рождения берет свое начало из теории вероятностей, в которой задача дня рождения имеет неочевидное решение. Сколько человек должно быть в комнате, чтобы с вероятностью не менее 50 % среди них оказались два человека с одним днем рождения (это и будет коллизией)? Оказывается, для этого достаточно 23 выбранных наугад человек! Странно, правда?

Это называется *парадоксом дней рождения*. На практике, когда мы случайным образом генерируем строки из пространства $2N$ возможностей, можно с вероятностью 50 % ожидать, что кто-то найдет коллизию после того, как сгенерирует примерно $2N/2$ строк.

Если наша хеш-функция генерирует случайные выходы размером 256 бит, то пространство всех выходов имеет размер 2^{256}. В таком случае высока вероятность нахождения коллизии после генерации 2^{128} дайджестов в связи с ограниченностью дней рождения. Именно к этому числу мы стремимся, и именно поэтому хеш-функции должны предоставлять как минимум 256-битные выходы.

Определенные ограничения иногда заставляют разработчиков уменьшать размер дайджеста путем *его усечения* (удаления части байтов), что теоретически возможно, но может значительно снизить уровень защиты. Для достижения как минимум 128-битной защиты дайджест должен быть не короче:

- 256 бит — для обеспечения устойчивости к коллизии;
- 128 бит — для обеспечения устойчивости к атаке нахождения первого и второго прообразов.

Таким образом, в зависимости от того, на какое свойство безопасности опирается хеш-функция, ее выход может быть усечен для получения более короткого дайджеста.

2.4. ХЕШ-ФУНКЦИИ НА ПРАКТИКЕ

Как уже упоминалось, на практике хеш-функции редко используются сами по себе. Чаще всего при создании криптографического примитива или криптографического протокола их комбинируют с другими элементами. В книге мы рассмотрим множество примеров применения хеш-функций при создании более сложных объектов, а в данном разделе описано несколько различных способов применения хеш-функций в реальном мире.

2.4.1. Обязательства

Допустим, вы знаете, что в ближайший месяц акции на рынке вырастут в цене до 50 долларов, но не можете рассказать об этом друзьям (возможно, по каким-то юридическим причинам). Однако вам хочется рассказать им о том, что вы узнали об этом росте заранее, и тем самым потешить свое самолюбие (не отрицайте этого). Что вы можете сделать, так это составить предложение типа: «Акции X достигнут 50 долларов в следующем месяце», захешировать его и отдать дайджест своим друзьям. Через месяц, когда вы раскроете предложение, они смогут захешировать его и убедиться в идентичности дайджестов.

Это то, что мы называем *схемой обязательств*. Как правило, обязательства в криптографии пытаются удовлетворять двум свойствам, вот они:

- *сокрытие* — обязательство должно скрывать базовое значение;
- *связывание* — обязательство должно скрывать одно значение. Другими словами, зафиксировав значение x, вы не должны иметь возможность впоследствии изменить это значение на y.

УПРАЖНЕНИЕ

Можете ли вы сказать, обеспечивает ли хеш-функция сокрытие и связывание, если она используется в качестве схемы обязательств?

2.4.2. Целостность субресурсов

Часто веб-страницы импортируют внешние файлы JavaScript. Например, многие веб-сайты используют сети доставки содержимого (Content Delivery Network, CDN) для импорта библиотек JavaScript или связанных с веб-фреймворками файлов на свои страницы. Такие CDN размещаются в стратегически важных местах, чтобы быстро доставлять эти файлы посетителям веб-страницы. Тем не менее CDN станет реальной проблемой, если выйдет из-под контроля и решит обслуживать вредоносные файлы JavaScript. Веб-страницы могут противостоять данной угрозе посредством функции, называемой *целостностью субресурсов*, которая позволяет включить дайджест в тег импорта:

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"
    integrity="sha256-8WqyJLuWKRBVhxXIL1jBDD7SDxU936oZkCnxQbWwJVw="></script>
```

Это тот же пример, который мы рассматривали во введении к этой главе. Получив файл JavaScript, браузер хеширует его с помощью SHA-256 и проверяет, совпадает ли он с дайджестом, встроенным в код страницы. Если совпадение подтверждается, файл выполняется, поскольку его целостность была проверена.

2.4.3. BitTorrent

Пользователи, называемые *пирами*, по всему миру с помощью протокола BitTorrent обмениваются файлами между собой, что мы называем *пирингом*. Распространяемый файл разрезается на фрагменты, каждый из которых затем индивидуально хешируется. Получившиеся хеши используются в качестве источника доверия к загружаемому файлу.

BitTorrent имеет несколько механизмов, позволяющих пирам получать различные фрагменты файла от различных пиров. В конечном счете целостность всего файла проверяется путем хеширования каждого загружаемого фрагмента и сопоставления выхода с известными дайджестами перед сборкой фрагментов в один файл. Например, следующая «магнет-ссылка»:

```
magnet:?xt=urn:btih:b7b0fbab74a85d4ac170662c645982a862826455
```

представляет операционную систему Ubuntu v19.04. Она является дайджестом (в шестнадцатеричном формате), который состоит из захешированных метаданных о файле и дайджестов всех его фрагментов.

2.4.4. Tor

Задача браузера Tor — предоставить пользователям возможность анонимно просматривать интернет-страницы. Другая его функция заключается в том, что он позволяет создавать скрытые веб-страницы, физическое местоположение которых сложно отследить. Соединения с этими страницами защищены протоколом, использующим открытый ключ веб-страницы. (Мы подробнее рассмотрим его работу в главе 9, когда будем говорить о шифровании сессий.) Например, Silk Road, который был своеобразным eBay по продаже наркотиков, до своей ликвидации ФБР был доступен в браузере Tor по адресу `silkroad6ownowfk.onion`. Эта строка base32 представляла собой хеш открытого ключа Silk Road. Таким образом, зная «луковый»[1] адрес, вы можете проверить подлинность открытого ключа скрытой веб-страницы, которую посещаете, и удостовериться в том, что общаетесь с правильной страницей, а не с самозванкой. Не волнуйтесь, если чего-то не поняли, мы вернемся к этой теме в главе 9.

УПРАЖНЕНИЕ

Кстати, такая строка никак не может представлять 256 бит (32 байта), верно? Как же тогда считать ее безопасной, если основываться на информации из раздела 2.3? Можете ли вы предположить, каким образом Dread Pirate Roberts (псевдоним администратора Silk Road) удалось получить хеш, содержащий название веб-сайта?

Во всех примерах в этом разделе хеш-функция обеспечивала *целостность* или *подлинность содержимого* в ситуациях, когда:

[1] Onion (англ.) — «лук». — *Примеч. пер.*

- кто-то мог подделать хешируемое содержимое;
- передача хеша происходила в защищенном режиме.

Иногда мы также говорим, что *проверяем аутентичность* чего-либо или кого-либо. Важно понимать, что если хеш был получен не в защищенном режиме, то любой может заменить его хешем чего-то другого! Таким образом, сам по себе хеш не обеспечивает целостности. Данная уязвимость исправляется использованием *секретов*, о чем будет рассказано в следующей главе, посвященной имитовставке. А теперь рассмотрим, какие алгоритмы хеш-функций можно применять.

2.5. СТАНДАРТИЗИРОВАННЫЕ ХЕШ-ФУНКЦИИ

В предыдущем примере я упоминал алгоритм SHA-256, который является лишь одной из хеш-функций, которые можно использовать. Прежде чем перейти к списку рекомендуемых современных хеш-функций, скажу пару слов об алгоритмах, которые применяются в реальных приложениях, но не считаются криптографическими хеш-функциями.

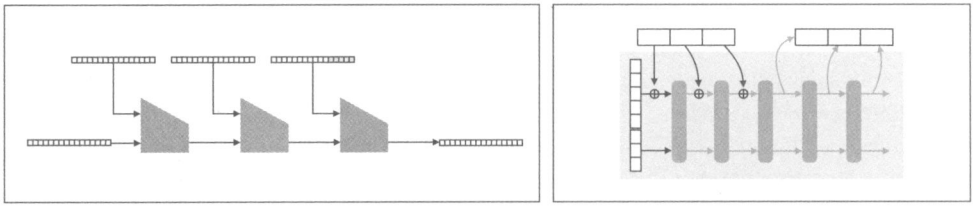
Во-первых, функции типа CRC32 — это *не* криптографические хеш-функции, а функции с кодом обнаружения ошибок. Они помогают найти некоторые простые ошибки, однако не удовлетворяют ни одному из ранее упомянутых свойств безопасности, поэтому их не следует путать с хеш-функциями, даже если они имеют одинаковое название. Выход таких функций обычно называют *контрольной суммой*.

Во-вторых, популярные хеш-функции MD5 и SHA-1 сегодня считаются ненадежными. MD5 и SHA-1 являлись общепризнанными стандартами в 1990-х годах, однако в 2004 и 2016 годах соответственно различные исследовательские группы опубликовали данные о найденных в них коллизиях. Успех атак на них отчасти обусловлен достижениями в области вычислительной техники, но основной причиной стали обнаруженные недостатки в дизайне этих хеш-функций.

ДЕПРЕКАЦИЯ БЕЗЖАЛОСТНА

И MD5, и SHA-1 считались хорошими хеш-функциями, пока исследователи не продемонстрировали их недостаточную устойчивость к коллизиям. На сегодняшний день не было произведено ни одной проверки их устойчивости к атаке нахождения первого и второго прообразов. Для нас это не имеет значения, поскольку здесь мы будем говорить только о безопасных алгоритмах. Тем не менее еще можно увидеть MD5 и SHA-1 в системах, где упор делается только на устойчивость этих алгоритмов к атакам нахождения прообразов, но не на устойчивость к коллизиям. В таких случаях часто утверждают, что невозможно заменить эти хеш-функции на более безопасные по причине устаревшего программного обеспечения и обратной совместимости. Поскольку эта книга призвана прослужить долго и стать лучом света для будущего реальной криптографии, я в последний раз упоминаю MD5 и SHA-1.

В следующих двух разделах представлены две наиболее широко используемые хеш-функции — SHA-2 и SHA-3 (рис. 2.6).



SHA-2 (Меркл — Дамгард) SHA-3 (губка)

Рис. 2.6. SHA-2 и SHA-3 — две наиболее распространенные хеш-функции. SHA-2 основана на структуре Меркла — Дамгарда, а SHA-3 — на функции губки

2.5.1. Хеш-функция SHA-2

Теперь, когда вы знаете, что такое хеш-функции, и познакомились с потенциальными вариантами их применения, остается выяснить, какие из них мы можем использовать на практике. В этом и следующем разделах я представлю две широко распространенные хеш-функции, а также в общих чертах объясню принцип их работы. Такое обобщенное объяснение не призвано обеспечить глубокое понимание того, как использовать хеш-функции, поскольку данной мной характеристики по принципу черного ящика должно быть достаточно. Тем не менее интересно посмотреть, как эти криптографические примитивы разрабатывались.

Наиболее широко распространенной хеш-функцией является *алгоритм безопасного хеширования — 2* (Secure Hash Algorithm 2, SHA-2). SHA-2 изобретен АНБ и стандартизирован NIST в 2001 году. Он разработан как дополнение к устаревающему алгоритму безопасного хеширования — 1 (SHA-1), уже стандартизированному NIST. SHA-2 имеет четыре версии, дающие на выходе 224, 256, 384 или 512 бит. В их названиях версия алгоритма не указывается: SHA-224, SHA-256, SHA-384 и SHA-512. Кроме того, есть еще две версии, SHA-512/224 и SHA-512/256, которые обеспечивают 224- и 256-битный выходы соответственно путем усечения результата SHA-512.

В следующей терминальной сессии мы вызовем каждый из вариантов SHA-2 с помощью командного интерфейса OpenSSL. Обратите внимание на то, что разные варианты преобразуют одни и те же входные данные в совершенно разные выходы заданной длины:

```
$ echo -n "hello world" | openssl dgst -sha224
2f05477fc24bb4faefd86517156dafdecec45b8ad3cf2522a563582b
$ echo -n "hello world" | openssl dgst -sha256
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
$ echo -n "hello world" | openssl dgst -sha384
fdbd8e75a67f29f701a4e040385e2e23986303ea10239211af907fcbb83578b3
➥ e417cb71ce646efd0819dd8c088de1bd
$ echo -n "hello world" | openssl dgst -sha512
309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f
➥ 989dd35bc5ff499670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f
```

В настоящее время в основном используется SHA-256, который обеспечивает минимальные 128 бит защиты, необходимые для того, чтобы удовлетворять трем свойствам безопасности, в то время как более параноидальные приложения применяют SHA-512. Теперь посмотрим на упрощенное объяснение работы SHA-2.

ИСКЛЮЧАЮЩЕЕ ИЛИ

Чтобы понять следующую информацию, необходимо иметь представление об операции XOR (исключающее ИЛИ). XOR — это битовая операция, то есть она оперирует битами. На следующем рисунке показано, как она работает. Операция XOR повсеместно используется в криптографии, поэтому обязательно запомните ее.

XOR

```
1 ⊕ 0 = 1
1 ⊕ 1 = 0
0 ⊕ 1 = 1
0 ⊕ 0 = 0
```

Исключающее ИЛИ, или XOR (часто обозначается символом ⊕), оперирует двумя битами. Она похожа на операцию ИЛИ (OR), кроме того случая, когда оба операнда равны 1.

Все начинается со специальной функции, называемой *функцией сжатия*. Она принимает два входных блока данных некоторого размера и производит один выход размером, равным размеру одного из этих блоков. Проще говоря, принимаются некоторые данные и возвращаются данные меньшего размера (рис. 2.7).



Рис. 2.7. Функция сжатия принимает два входных блока данных размером X и Y (здесь оба по 16 байт) и возвращает выход размером либо X, либо Y

Существуют различные способы построения функции сжатия. В SHA-2 используется метод *Дэвиса — Мейера* (рис. 2.8), который основывается на *блочном шифре*, то есть на шифровании блока данных фиксированного размера. Я упоминал блочный шифр AES в главе 1, но подробно мы его не рассматривали. Воспринимайте функцию сжатия как черный ящик. Когда прочтете в главе 4 о шифровании с проверкой подлинности, эта функция станет для вас более понятной.



Рис. 2.8. Изображение функции сжатия, построенной по структуре Дэвиса — Мейера. Первые данные на входе (входной блок данных) используются в качестве ключа для блочного шифра. Вторые данные на входе (промежуточное значение) предназначены для шифрования блочным шифром. Затем над этими же данными выполняется операция XOR, в результате чего и получается выход блочного шифра

SHA-2 является структурой *Меркла — Дамгарда*, то есть это алгоритм (изобретен Ральфом Мерклом и Иваном Дамгардом независимо друг от друга), который хеширует сообщение путем итеративного вызова функции сжатия. Как правило, он состоит из следующих двух шагов.

Сначала к хешируемым входным данным применяется *дополнение* (padding). Затем входные данные разрезаются на блоки, помещающиеся в функцию сжатия. Под дополнением подразумевается добавление определенного количества байтов к входным данным, чтобы сделать их длину кратной размеру блока. Разбивка дополненных входных данных на блоки одинакового размера позволяет поместить их в первый аргумент функции сжатия. Например, SHA-256 имеет размер блока 512 бит. Этот шаг представлен на рис. 2.9.

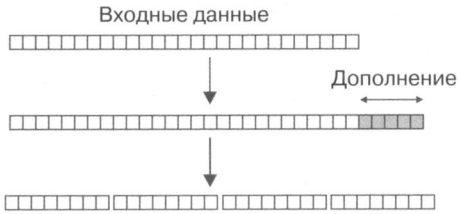


Рис. 2.9. Первым шагом в структуре Меркла — Дамгарда является дополнение входного сообщения некоторой информацией, в результате которого его длина должна стать кратной размеру входа используемой функции сжатия (например, 8 байт). Для этого мы добавляем 5 байт дополнительной информации к концу сообщения, чтобы его длина достигла 32 байт. Затем разрезаем сообщение на четыре блока по 8 байт

Далее к полученным блокам итеративно применяется функция сжатия. При этом выход предыдущей функции задействуется в качестве второго аргумента для следующей. Конечным результатом является *дайджест*. Этот шаг представлен на рис. 2.10.

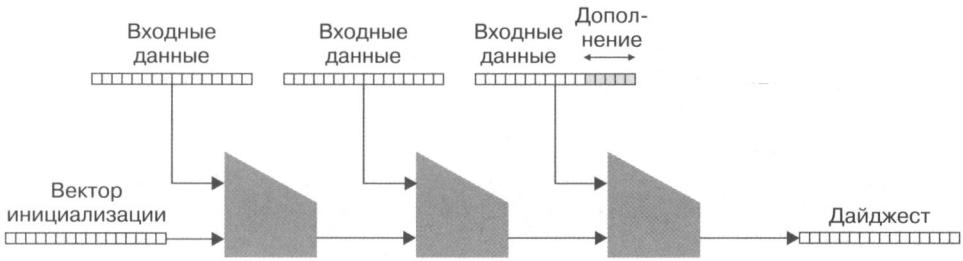


Рис. 2.10. Структура Меркла — Дамгарда итеративно применяет функцию сжатия к каждому блоку хешируемых входных данных с использованием выхода предыдущей функции сжатия. Последний вызов функции возвращает непосредственно дайджест

Именно так работает SHA-2: она итеративно использует функцию сжатия на фрагментах входных данных, пока все не будет преобразовано в окончательный дайджест.

ПРИМЕЧАНИЕ

Структура Меркла – Дамгарда устойчива к коллизиям, если устойчива сама функция сжатия. Таким образом, безопасность хеш-функции, применяемой к *входным данным произвольной длины*, сводится к безопасности функции сжатия *фиксированного* размера, которую легче разработать и проанализировать. В этом и заключается изобретательность структуры Меркла – Дамгарда.

С самого начала второй аргумент функции сжатия, как правило, закреплен и стандартизирован значением *nothing-up-my-sleeve* (дословно — «в рукавах ничего нет»). SHA-256 получает его с помощью квадратных корней первых простых чисел. Значение nothing-up-my-sleeve призвано убедить криптографическое сообщество в том, что оно выбрано не для того, чтобы сделать хеш-функцию слабее (например, для создания черного хода). В криптографии данная концепция пользуется большой популярностью.

ВНИМАНИЕ

Хотя SHA-2 – прекрасная хеш-функция, она не подходит для хеширования секретов. У структуры Меркла – Дамгарда есть недостаток, который делает SHA-2 уязвимой для *атаки удлинением сообщения* при использовании ее для хеширования секретов. Более подробно поговорим об этом в следующей главе.

2.5.2. Хеш-функция SHA-3

Как было упомянуто ранее, совсем недавно хеш-функции MD5 и SHA-1 были взломаны. Они обе использовали структуру Меркла – Дамгарда, которую я описал в предыдущем разделе. В связи с этим, а также с тем, что SHA-2 уязвима к атаке удлинением сообщения, в 2007 году NIST решил провести открытый конкурс на новый стандарт – *SHA-3*. В этом разделе речь пойдет именно об этой хеш-функции, и я попытаюсь в общих чертах объяснить ее внутреннюю работу.

В 2007 году в конкурсе SHA-3 приняли участие 64 алгоритма, созданных различными международными исследовательскими группами. Пять лет спустя один из них, Keccak, был объявлен победителем и стал SHA-3. В 2015 году FIPS Publication 202 (https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf) утвердил SHA-3 в качестве стандарта.

SHA-3 удовлетворяет трем свойствам безопасности, о которых мы говорили ранее, и обеспечивает такую же защиту, как и различные версии SHA-2. Кроме того, она не подвержена атакам удлинением сообщения и может применяться для хеширования секретов. По этой причине сейчас рекомендуется использовать именно эту хеш-функцию. SHA-3 предлагает те же варианты, что и SHA-2, но уже с указанием полного названия: SHA-3-224, SHA-3-256, SHA-3-384 и SHA-3-512. Таким образом, аналогично SHA-2, SHA-3-256 обеспечивает, например, 256 бит на выходе. Теперь выделим несколько страниц на то, чтобы объяснить, как работает эта хеш-функция.

SHA-3 — это криптографический алгоритм, построенный на основе *перестановки*. Самый простой способ понять *перестановку* — это представить, что у вас есть набор элементов слева и такой же набор элементов справа. Соедините эти элементы стрелками слева направо. К каждому элементу и от каждого элемента должна быть проведена только одна стрелка. Это и есть перестановка. На рис. 2.11 показан этот принцип работы. По определению любая перестановка *обратима*, то есть по выходу мы можем найти входные данные.



Рис. 2.11. Пример перестановки четырех фигур. Перестановка, изображенная стрелками на среднем рисунке, может быть использована для преобразования заданной фигуры

Алгоритм SHA-3 имеет *конструкцию губки*. Она отличается от структуры Меркла — Дамгарда (также представленной на конкурсе SHA-3) тем, что основывается на определенной перестановке под названием *keccak-f*, которая принимает входные данные и возвращает выход того же размера.

ПРИМЕЧАНИЕ

Я не стану рассказывать, как был создан keccak-f, но в главе 4 вы получите о нем представление, так как он очень похож на алгоритм AES (их различие в том, что у последнего есть ключ). Это не совпадение, поскольку один из изобретателей AES разрабатывал SHA-3.

На следующих нескольких страницах я буду использовать восьмибитную перестановку в качестве примера работы функции губки. Поскольку перестановку нельзя изменить, рис. 2.12 хорошо иллюстрирует преобразование, созданное этой перестановкой на всех возможных восьмибитных входных данных. На основании предыдущего объяснения перестановки можно также представить, что каждая возможная восьмибитная строка имеет вид какой-то фигуры (`000...` — треугольник, `100...` — квадрат и т. д.).

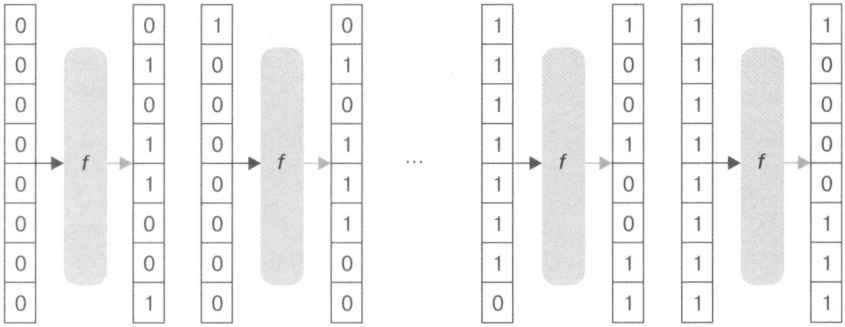


Рис. 2.12. Конструкция губки задействует заданную перестановку f. Оперируя входными данными, перестановка обеспечивает преобразование всех возможных восьмибитных входных данных во все возможные восьмибитные выходы

При использовании перестановки в конструкции губки необходимо определить произвольное разделение входных данных и выхода на *скорость* (rate) и *емкость* (capacity). Этот процесс (рис. 2.13) немного непонятен, но его нельзя игнорировать.

Граница между скоростью и емкостью устанавливается произвольно. В разных версиях SHA-3 используются разные параметры. Если сказать проще, емкость следует рассматривать как секрет, и чем она больше, тем надежнее конструкция губки.

Теперь, как и в случае со всеми хорошими хеш-функциями, мы должны иметь возможность что-то хешировать, верно? Иначе все изложенное не имеет смысла. Для этого мы просто выполняем операцию XOR (⊕) над входными данными, отнесенными к скорости перестановки входных данных. Сначала эта часть является просто кучей нулей. Как мы отмечали ранее, емкость рассматривается как секрет, поэтому не будем выполнять операцию XOR над этой частью входных данных (рис. 2.14).

Полученный результат теперь должен иметь случайный вид, хотя мы без труда можем определить входные данные, поскольку перестановка обратима по определению. А что, если мы хотим поглотить входные данные подлиннее? Как и в ходе работы с SHA-2, делаем следующее.

1. При необходимости дополняем входные данные, а затем делим их на блоки нужного размера.
2. Над каждым блоком с входными данными перестановки выполняем операцию XOR, после чего к *состоянию* (промежуточному результату последней операции) применяем перестановку.

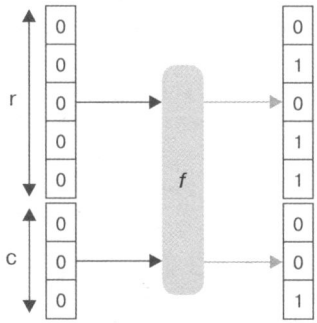


Рис. 2.13. Перестановка f случайным образом преобразует входные данные размером 8 бит в выход того же размера. В конструкции губки входные и выходные данные этой перестановки делятся на две части: скорость (размером r) и емкость (размером c)

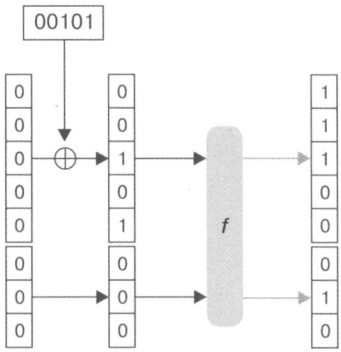


Рис. 2.14. Для впитывания 5 бит входных данных 00101 конструкция губки, содержащая 5 бит данных, являющихся скоростью, может просто выполнить операцию XOR над этими 5 битами данных (она инициализируется нулями). Затем перестановка случайным образом преобразует состояние

Чтобы не перегружать дальнейшее объяснение, я больше не буду упоминать о дополнении входных данных. Тем не менее следует помнить, что дополнение — важный этап процесса хеширования, позволяющий различать входные данные, например `0` и `00`. На рис. 2.15 показаны эти два этапа.

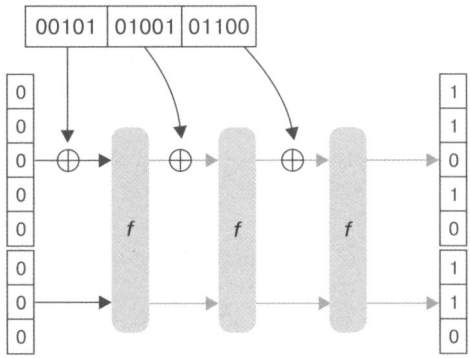


Рис. 2.15. Чтобы впитать входные данные, длина которых больше размера скорости, конструкция губки итеративно выполняет операцию XOR над блоками входных данных, являющихся скоростью, и переставляет результат

Пока все хорошо, но у нас все еще нет дайджеста. Для его получения можем просто использовать скорость последнего состояния губки (опять же не трогаем емкость). Чтобы дайджест был длиннее, можем продолжить перестановку отнесенной к скорости части состояния (рис. 2.16).

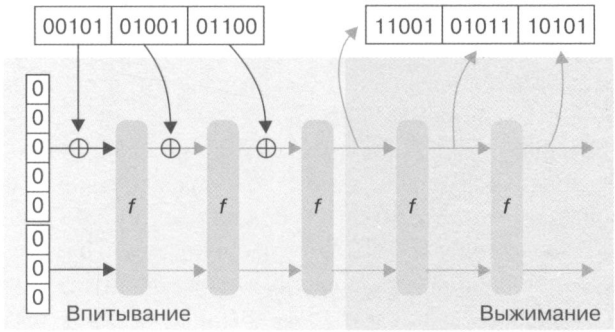


Рис. 2.16. Чтобы получить дайджест с помощью конструкции губки, нужно итеративно переставлять состояние и забирать требуемое количество скорости (верхней части состояния)

Именно так работает SHA-3. Поскольку она имеет *конструкцию губки*, поглощение входных данных, естественно, называется *впитыванием*, а создание дайджеста — *выжиманием*. Губка задается 1600-битной перестановкой с использованием различных значений r и c в зависимости от защиты, обеспечиваемой разными версиями SHA-3.

SHA-3 — ЭТО СЛУЧАЙНЫЙ ОРАКУЛ

Ранее я рассказывал о случайном оракуле — вымышленной идеальной конструкции, которая возвращает совершенно случайные ответы на запросы и повторяет себя, если мы дважды делаем одинаковый запрос. Оказалось, что конструкция губки действует почти так же, как этот случайный оракул, если используемая в ней перестановка выглядит достаточно случайной. Как доказать случайность? Лучший подход заключается в том, чтобы многократно пытаться сломать конструкцию, пока не будет получена твердая уверенность в ее надежности (что и произошло во время конкурса по SHA-3). Тот факт, что SHA-3 может быть смоделирована как случайный оракул, позволил говорить, что она обладает теми свойствами безопасности, которых мы ожидаем от хеш-функции.

2.5.3. SHAKE и cSHAKE — две функции расширенного выхода (XOF)

Я представил два основных стандарта хеш-функций — SHA-2 и SHA-3. Это четко прописанные хеш-функции, которые принимают входные данные произвольной длины и на выходе выдают данные произвольного вида и фиксированной длины. Как вы увидите в последующих главах, криптографические протоколы часто требуют примитивов такого типа, но не хотят ограничиваться фиксированными

размерами дайджеста. По этой причине в стандарт SHA-3 был введен более универсальный примитив, называемый *функцией расширенного выхода* (extendable output function), или *XOF* (произносится «зоф»). В этом разделе представлены две стандартизированные XOF – SHAKE и cSHAKE.

SHAKE, указанная в FIPS 202 наряду с SHA-3, может рассматриваться как хеш-функция, которая возвращает выход произвольной длины. Это, по сути, та же конструкция, что и SHA-3, за исключением того, что она быстрее и на этапе выжимания выполняет столько перестановок, сколько нужно. Выходные данные разного размера весьма полезны не только для создания дайджеста, но и для генерации случайных чисел, получения ключей и т. д. В книге я еще не раз буду говорить о различных применениях SHAKE, а пока ограничимся тем, что SHAKE похожа на SHA-3, за исключением того, что она предоставляет выход любой нужной длины.

В области криптографии эта конструкция оказалась такой успешной, что через год после стандартизации SHA-3 NIST выпустил специальную публикацию 800-185, в которой представил *настраиваемую версию SHAKE* под названием *cSHAKE* (customizable SHAKE). Она практически аналогична SHAKE, за исключением того, что вместе с входными данными принимает строку настройки. Эта строка может быть пустой, а может содержать любую настройку. Давайте сначала рассмотрим пример использования cSHAKE в псевдокоде:

```
cSHAKE(input="hello world", output_length=256, custom_string="мой_хеш")
-> 72444fde79690f0cac19e866d7e6505c
cSHAKE(input="hello world", output_length=256, custom_string="твой_хеш")
-> 688a49e8c2a1e1ab4e78f887c1c73957
```

Здесь можно увидеть различие двух дайджестов, несмотря на то что cSHAKE также детерминирована, как SHAKE и SHA-3. Это связано с тем, что во втором случае была применена другая строка настройки. *Строка настройки* (customization string) позволяет настраивать XOF! Она полезна в некоторых протоколах, где, например, для того, чтобы доказательство работало, необходимо использовать разные хеш-функции. Мы называем это *разделением предметных областей* (domain separation).

Золотое правило криптографии гласит: если один и тот же криптографический примитив используется в разных сценариях, не применяйте его с одним и тем же ключом (если требуется ключ) и/или разделяйте предметные области. Вы увидите больше примеров разделения предметных областей в последующих главах.

ВНИМАНИЕ

Как правило, NIST указывает алгоритмы, которые принимают параметры в битах, а не в байтах. В приведенном ранее примере была запрошена длина 256 бит. Представьте, что вы бы запросили длину 16 байт, а получили 2 байта, потому что программа подумала, что вы запросили 16 бит выхода. Эту проблему иногда называют *битовой атакой*.

Как и все в криптографии, длина таких криптографических строк, как ключи, параметры и выходные данные, тесно связана с безопасностью системы. Важно

избегать запроса слишком коротких выходных данных у SHAKE или cSHAKE. *Невозможно ошибиться, используя выход длиной 256 бит*, поскольку он обеспечивает 128 бит защиты от атак нахождения коллизий. Но реальная криптография иногда работает в стесненных условиях, которые позволяют применять более короткие криптографические значения. Это допустимо, если тщательно проанализировать безопасность системы. Например, если устойчивость к коллизиям неважна в протоколе, использующем значение, для устойчивости к атаке нахождения прообраза достаточно выходов SHAKE или cSHAKE длиной 128 бит.

2.5.4. Избегание неоднозначного хеширования с помощью TupleHash

Я уже рассказал о различных криптографических примитивах и алгоритмах:

- хеш-функции SHA-2, уязвимой к атакам удлинением сообщения, но все еще широко используемой, когда не нужно хешировать никакие секреты;
- хеш-функции SHA-3, которая в настоящее время является рекомендуемой хеш-функцией;
- XOF-функциях SHAKE и cSHAKE – более универсальных инструментах, чем хеш-функции, поскольку предлагают вариативную длину выхода.

Есть еще одна удобная функция, *TupleHash*. Она основана на cSHAKE и определена в том же стандарте, что и cSHAKE. Это интересная функция, которая позволяет хешировать *кортеж* (список чего-либо). Чтобы объяснить, что такое TupleHash и почему она полезна, позвольте мне рассказать одну историю.

Несколько лет назад я получил на работе задание выполнить обзор одной криптовалюты. Она включала в себя основные функции, которые можно было ожидать: счета, платежи и т. д. Транзакции между пользователями содержали метаданные о том, кто сколько кому отправляет. Кроме того, взималась небольшая плата, чтобы компенсировать сети обработку транзакции.

Алиса, например, может отправлять транзакции в сеть, но для того, чтобы они были приняты, ей необходимо доказать, что транзакция исходит от нее. Для этого она может захешировать транзакцию и подписать ее (я приводил аналогичный пример в главе 1). Любой может захешировать транзакцию и проверить подпись на хеше, чтобы убедиться, что Алиса хотела отправить именно эту транзакцию. На рис. 2.17 показано, что атакующий посредник, перехвативший транзакцию до ее попадания в сеть, не сможет ее подделать. Это связано с тем, что при вмешательстве хеш изменится и подпись не подтвердит подлинность нового дайджеста транзакции.

В главе 7 вы увидите, что такой злоумышленник, конечно же, не сможет подделать подпись Алисы на новом дайджесте. Благодаря устойчивости используемой хеш-функции к нахождению второго прообраза он также не сможет найти другую транзакцию, которая после хеширования имела бы тот же самый дайджест.

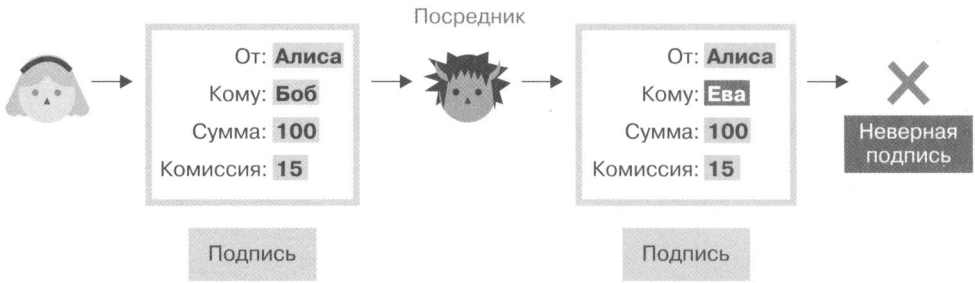


Рис. 2.17. Алиса отправляет транзакцию, а также подпись к хешу транзакции. Если атакующий посредник попытается подделать транзакцию, хеш будет другим и, следовательно, прилагаемая подпись окажется неверной

Значит, посредник безвреден? Нам еще рано расслабляться. В случае с криптовалютой, которую я проверял, транзакция хешировалась простым объединением каждого поля:

```
$ echo -n "Алиса""Боб""100""15" | openssl dgst -sha3-256
06618880ab31274668a9e020e467f73b091c4f563ac6f238b2797bba2d9b8710
```

То, что выглядело совершенно нормальным, на самом деле полностью ломало платежную систему криптовалюты. Такой хеш тривиально позволяет злоумышленнику найти второй прообраз хеш-функции. Подумайте, как можно найти другую транзакцию, которая после хеширования имела бы тот же дайджест `0661...`.

Что произойдет, если мы перенесем одну цифру из поля «Комиссия» в поле «Сумма»? Можно увидеть, что следующая транзакция хешируется в тот же дайджест, который подписала Алиса:

```
$ echo -n "Алиса""Боб""1001""5" | openssl dgst -sha3-256
06618880ab31274668a9e020e467f73b091c4f563ac6f238b2797bba2d9b8710
```

Таким образом, посредник, который захочет, чтобы Боб получил немного больше денег, сможет изменить транзакцию с сохранением валидности подписи. Как вы уже, наверное, догадались, именно эту проблему решает TupleHash. Данная функция позволяет однозначно хешировать список полей, используя недвусмысленное кодирование. В реальности происходит нечто близкое к следующему (с операцией объединения строк `||`):

```
cSHAKE(input="5"||"Алиса"||"3"||"Боб"||"3"||"100"||"2"||"10",
➥ output_length=256, custom_string="TupleHash"+"все, что нужно")
```

На этот раз входные данные строятся путем указания длины в начале каждого поля транзакции. Здесь можно остановиться на минутку и поразмышлять, почему это решает нашу проблему. В целом можно безопасно использовать любую хеш-функцию, если входные данные *сериализуются* перед хешированием. Сериализация входных данных означает, что всегда существует способ их *десериализации*, то есть восстановления исходных данных. Если данные можно десериализовать, то нет никакой неоднозначности в разграничении полей.

2.6. ХЕШИРОВАНИЕ ПАРОЛЕЙ

В этой главе вы видели несколько полезных функций, которые либо являются хеш-функциями, либо расширяют возможности существующих хеш-функций. Но прежде, чем вы сможете перейти к следующей главе, я должен упомянуть о *хешировании паролей*.

Представьте следующий сценарий: у вас есть сайт (то есть вы его администратор) и вам нужно, чтобы пользователи регистрировались на нем и затем входили в свои учетные записи. По этой причине вы создаете две веб-страницы для этих двух функций. Внезапно возникает вопрос: как вы собираетесь хранить пароли? Они будут храниться в открытом виде в базе данных? Сначала кажется, что в этом нет ничего плохого. Однако это решение имеет слабое место. Люди склонны повсеместно применять один и тот же пароль, и если (или когда) вас взломают и злоумышленникам удастся получить все пользовательские пароли, это негативно скажется на безопасности пользователей и репутации вашей платформы. Еще немного поразмышляв, вы понимаете, что злоумышленник, укравший такую базу данных, сможет войти в систему под именем любого пользователя. Таким образом, хранение паролей в открытом виде становится идеей, далекой от идеала, и нужно найти лучший способ решения этой проблемы.

Одним из решений может быть хеширование паролей и хранение только дайджестов. Когда кто-то входит на ваш сайт, поток данных будет выглядеть следующим образом.

1. Вы получаете пароль пользователя.
2. Вы хешируете введенный пароль и удаляете его.
3. Вы сравниваете получившийся дайджест с тем, что хранится в вашей базе: если он совпадает, пользователь входит в учетную запись.

Этот поток позволяет вам обрабатывать пароли пользователей в течение ограниченного времени. Тем не менее злоумышленник, проникший на ваши серверы, до того, как будет обнаружен, может незаметно записывать пароли из этого потока. Необходимо признать, что это тоже неидеальное решение, но все же мы повысили безопасность сайта. В сфере обеспечения безопасности данный подход называется защитой в глубину, то есть наслоением несовершенных средств обеспечения безопасности в надежде, что злоумышленник не сможет преодолеть все слои. Именно этим и занимается криптография в реальном мире. Однако у данного решения имеются и другие проблемы.

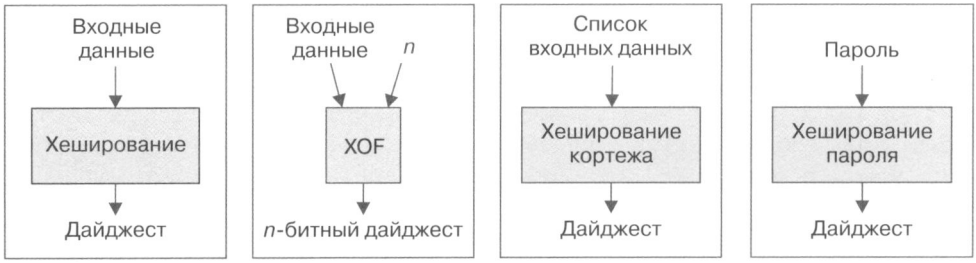
- *Если злоумышленник получит хешированные пароли, он может произвести атаку грубой силы или исчерпывающий поиск (перебор всех возможных паролей).* При этом каждая попытка поиска соответствий будет направлена на всю базу данных. Было бы идеально, если бы злоумышленник мог атаковать только один хешированный пароль за раз.
- *Хеш-функции должны быть быстрыми.* Злоумышленники могут использовать высокую скорость для перебора (много-много паролей в секунду). В идеале мы должны иметь механизм, замедляющий такие атаки.

Первая проблема обычно решается с помощью *соли*, представляющей собой открытые случайные значения, которые различаются для каждого пользователя. Соль применяется вместе с паролем пользователя при его хешировании, что в некотором смысле похоже на применение строки настройки для каждого пользователя в cSHAKE: она эффективно создает отдельную хеш-функцию для каждого пользователя. Поскольку для каждого из них задействуется своя хеш-функция, злоумышленник не может заранее составить большие таблицы паролей (так называемые *радужные таблицы*) для того, чтобы затем сравнить их с базой данных украденных хешей паролей.

Вторая проблема решается с помощью *хешей паролей*, которые разработаны так, чтобы быть медленными. В настоящее время для этого используется *Argon2* — победитель конкурса хеширования паролей (https://passwordhashing.net), проводимого с 2013 по 2015 год. На момент написания книги (2021 год) Argon2 находится на пути к стандартизации как RFC (https://datatracker.ietf.org/doc/draftirtfcfrgargon2/). На практике применяются и другие нестандартизированные алгоритмы, такие как PBKDF2, bcrypt и scrypt. Проблема здесь в том, что они могут использоваться с небезопасными параметрами и, следовательно, на практике их не так просто настроить.

Кроме того, только Argon2 и scrypt защищают от высокого уровня оптимизации со стороны злоумышленников, поскольку другие схемы не зависят от памяти. Термин *«зависимость от памяти»* (memory hard) означает, что алгоритм может быть оптимизирован только за счет оптимизации доступа к памяти. Другими словами, оптимизация всего остального ничего не даст. Поскольку оптимизация доступа к памяти ограничена даже при использовании специализированного оборудования (мы имеем только то количество кэша, какое можем разместить вблизи процессора), зависимые от памяти функции выполняются медленно на любом типе устройств. Это желательное свойство в случае, когда нужно не позволить злоумышленникам достичь значительного преимущества в скорости при оценке функции.

На рис. 2.18 показаны различные типы хеш-функций, которые вы видели в этой главе.



1. SHA-2, SHA-3, K12, BLAKE3 | 2. SHAKE, cSHAKE | 3. TupleHash | 4. Argon2

Рис. 2.18. Знакомые вам четыре типа хеш-функций: *1* — обычные, которые предоставляют уникальный идентификатор для входных данных произвольной длины; *2* — функции расширенного выхода, они похожи на обычные, но дают выход произвольной длины; *3* — хеш-функции для кортежей, которые однозначно перечисляют хеш-значения; *4* — функции хеширования паролей, плохо поддающиеся оптимизации, для безопасного хранения паролей

РЕЗЮМЕ

- Хеш-функция обеспечивает устойчивость к коллизиям, а также к атакам нахождения первого и второго прообразов.
  - Устойчивость к атаке нахождения прообраза означает, что невозможно найти входные данные, на основе которых был получен дайджест.
  - Устойчивость к атаке нахождения второго прообраза означает, что из одного входного блока данных и его дайджеста невозможно найти другой входной блок данных, после хеширования которого получается тот же дайджест.
  - Устойчивость к коллизиям означает, что должно быть невозможно найти два случайных входных блока данных, которые после хеширования имеют один и тот же выход.
- Наиболее распространенная хеш-функция — SHA-2, но из-за ее недостаточной устойчивости к атакам удлинением сообщения рекомендуемой хеш-функцией является SHA-3.
- SHAKE — это функция расширенного выхода (XOF), которая действует как хеш-функция, но предоставляет дайджест произвольной длины.
- cSHAKE (настраиваемая версия SHAKE) позволяет легко создавать экземпляры SHAKE, которые действуют как различные XOF. Это называется разделением предметных областей.
- Объекты перед хешированием следует сериализовать, чтобы избежать нарушения устойчивости хеш-функции к атаке нахождения второго прообраза. В этом помогают автоматические алгоритмы типа TupleHash.
- Для хеширования паролей используются более медленные хеш-функции, разработанные специально для этой цели. Argon2 — самое передовое решение на сегодняшний день.

3 Имитовставки

https://t.me/it_boooks/2

В этой главе

- ✓ Имитовставки.
- ✓ Свойства безопасности и подводные камни имитовставок.
- ✓ Самые широко используемые стандарты имитовставок.

Если смешать хеш-функцию с секретным ключом, то получится так называемый *код аутентификации сообщения*, или *имитовставка*[1] (message authentication code, MAC) — криптографический примитив, используемый для защиты целостности данных. Добавление секретного ключа — это основа любого типа безопасности: без ключа нереально обеспечить ни конфиденциальность, ни подлинность. Хеш-функции могут обеспечить подлинность или целостность произвольных данных, но не сами по себе, а благодаря дополнительному доверенному каналу, который невозможно подделать. В этой главе мы рассмотрим, каким образом используется имитовставка для создания такого доверенного канала и что еще она может делать.

ПРИМЕЧАНИЕ

Чтобы лучше понять эту главу, рекомендую прочитать главу 2, посвященную хеш-функциям.

[1] В русскоязычном пространстве чаще употребляется именно термин «имитовставка», но нередко встречается аббревиатура MAC. — *Примеч. пер.*

3.1. ФАЙЛЫ COOKIE БЕЗ СОХРАНЕНИЯ СОСТОЯНИЯ: ПОЯСНЯЮЩИЙ ПРИМЕР ИМИТОВСТАВОК

Представьте следующий сценарий: вы — веб-страница. Вы яркая, красочная и особенно гордитесь тем, что обслуживаете сообщество лояльных пользователей. Для взаимодействия с вами посетители должны войти в систему, отправив вам свои учетные данные, которые нужно проверить. Если учетные данные совпадают с теми, которые были введены при регистрации, то *аутентификация* пользователя *проходит успешно*.

Разумеется, просмотр веб-страниц состоит не из одного, а из множества запросов. Чтобы пользователям не приходилось заново проходить аутентификацию при каждом запросе, их браузер может хранить учетные данные и автоматически отправлять их серверу в нужный момент. Для этого в браузерах предусмотрена функция *cookies*! Файлы cookie предназначены не только для учетных данных: в них можно хранить все, что пользователь должен отправлять при каждом запросе.

Несмотря на то что этот бесхитростный подход хорошо работает, хранить в браузере такую конфиденциальную информацию, как пароли пользователей, в открытом виде небезопасно. В сессионном файле cookie чаще всего хранится случайная строка, которая генерируется сразу после входа пользователя в систему. Веб-сервер хранит ее во временной базе данных вместе с его именем. Если браузер каким-либо образом раскрывает сессионный файл cookie, то утечки информации о пароле пользователя не будет, хотя злоумышленник может задействовать его, чтобы притвориться пользователем. Веб-сервер также имеет возможность завершить сессию, удалив cookie на своей стороне, что очень удобно.



В этом подходе нет ничего плохого, но в некоторых случаях он плохо масштабируется. При наличии большого количества серверов может быть неудобно передавать им всем связки пользователей и случайных строк. В качестве альтернативного варианта можно хранить больше информации на стороне браузера. Давайте посмотрим, как это сделать.

Если пойти по самому простому пути и указать в файле cookie имя пользователя вместо случайной строки, это, очевидно, приведет к проблеме, поскольку

теперь я могу вручную изменить это имя и тем самым выдать себя за любого пользователя. В таком случае, возможно, нам помогут хеш-функции из главы 2. Подумайте, как они могут помешать пользователю подделать свои собственные файлы cookie.

Второй простой подход заключается в том, чтобы хранить в cookie не только имя пользователя, но и его дайджест. Для хеширования имени пользователя можно применить хеш-функцию типа SHA-3 (рис. 3.1). Как вы думаете, это сработает?



Рис. 3.1. Для аутентификации запросов веб-сервер просит браузер сохранить имя пользователя и его хеш и отправлять эту информацию с каждым последующим запросом

Здесь возникает большая проблема. Не стоит забывать, что хеш-функция является публичным алгоритмом и может быть пересчитана злоумышленником на основе новых данных. Хеш не обеспечивает целостности данных, если нет доверия к его источнику! На рис. 3.2 показано, что если злоумышленник изменит имя пользователя в cookie, то он может просто пересчитать дайджест.

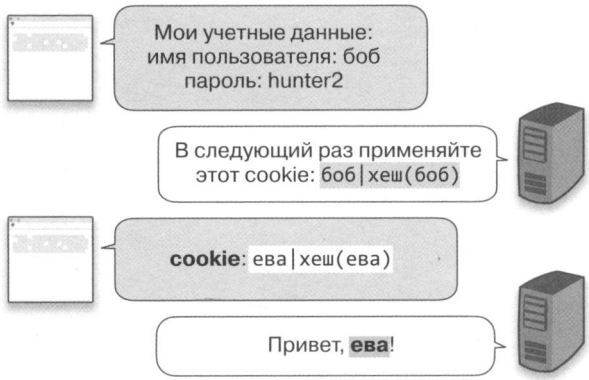


Рис. 3.2. Злоумышленник может изменить информацию, содержащуюся в его файлах cookie. Если cookie содержит имя пользователя и хеш, он может изменить оба эти параметра и выдать себя за другого пользователя

Тем не менее применение хеша нельзя назвать глупой идеей. Что еще мы можем сделать? Оказывается, существует аналогичный хеш-функции примитив — имитовставка (MAC), которая как раз и будет выполнять то, что нам нужно.

Имитовставкой называется алгоритм с использованием секретного ключа, который принимает не только входные данные, как хеш-функция, но и (кто бы мог подумать?) секретный ключ. Выходом MAC является уникальный *аутентификационный тег*. Данный процесс детерминирован: при одном и том же секретном ключе и одном и том же сообщении MAC выдает один и тот же аутентификационный тег (рис. 3.3).

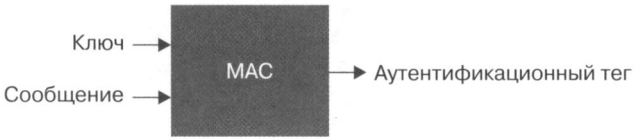


Рис. 3.3. Работа имитовставки. Алгоритм принимает секретный ключ и сообщение и детерминированно возвращает уникальный аутентификационный тег. Должно быть невозможно воспроизвести этот тег без ключа

Воспользуемся этим новым примитивом, чтобы убедиться, что пользователь не сможет подделать свой cookie. Когда он входит в систему в первый раз, из секретного ключа и его имени создается аутентификационный тег. Затем он должен сохранить свое имя пользователя и аутентификационный тег в файл cookie. Поскольку пользователь не знает секретного ключа, он не сможет создать корректный аутентификационный тег для другого имени пользователя.

Для подтверждения подлинности cookie вы делаете то же самое: создаете аутентификационный тег из секретного ключа и имени пользователя, находящегося в cookie, и проверяете, совпадает ли он с содержимым cookie. Если совпадает, значит, он точно ваш, так как только вы могли создать такой тег (со своим секретным ключом). На рис. 3.4 наглядно показан процесс проверки.

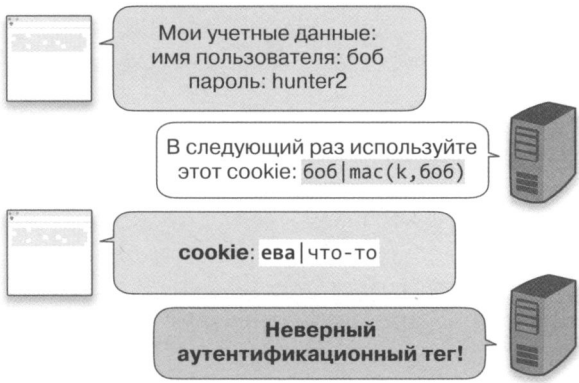


Рис. 3.4. Злоумышленник подделывает cookie, но не может создать актуальный аутентификационный тег для него. В результате веб-страница не может подтвердить подлинность и целостность нового cookie и, соответственно, отклоняет запрос

MAC — это как закрытая хеш-функция, которую можете вычислить только вы, поскольку знаете ключ. В некотором смысле с помощью ключа можно персонализировать хеш-функцию. Но этим связь MAC с хеш-функциями не ограничивается. Далее в этой главе мы увидим, что хеш-функции часто служат основой для построения имитовставок. Рассмотрим новый пример с использованием реального кода.

3.2. ПРИМЕР С КОДОМ

До этого момента только вы использовали MAC. Давайте в качестве стимула для написания кода увеличим число участников и посмотрим, как MAC применяются на практике. Представьте, что вы хотите общаться с человеком, при этом приватность переписки для вас неважна. Но что вас действительно волнует, так это целостность сообщений: они должны оставаться неизменными! Для этого вы и ваш собеседник должны использовать один и тот же секретный ключ вместе с имитовставкой.

В данном примере задействуем одну из наиболее популярных MAC-функций — *имитовставку, использующую хеш-функции* (hash-based message authentication code, HMAC), на языке программирования Rust. Другими словами, HMAC — это код аутентификации сообщений, в основе которого лежит хеш-функция. Он совместим с различными хеш-функциями, но чаще всего применяется в сочетании с SHA-2. Как показано в листинге 3.1, отправляющая сторона просто принимает ключ и сообщение и возвращает аутентификационный тег.

Листинг 3.1. Отправка аутентифицированного сообщения в Rust

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn send_message(key: &[u8], message: &[u8]) -> Vec<u8> {
    let mut mac = Hmac::<Sha256>::new(key.into());
    mac.update(message);
    mac.finalize().into_bytes().to_vec()
}
```

Создает объект HMAC с секретным ключом и хеш-функцией SHA-256

Запись большего количества входных данных в буфер для HMAC

Возвращает аутентификационный тег

Аналогичный процесс происходит и на стороне получателя. Получив сообщение и аутентификационный тег, ваш друг может сгенерировать собственный тег с тем же секретным ключом и сравнить их. Как и при шифровании, обе стороны должны использовать один и тот же секретный ключ. В листинге 3.2 показано, как это работает.

Обратите внимание на то, что этот протокол неидеален — он допускает повторы. Если сообщение и его аутентификационный тег будут воспроизведены позднее, они по-прежнему будут считаться подлинными, но у вас не будет возможности узнать, что вам снова отправили старое сообщение. Позже в этой главе я расскажу о решении данной проблемы. Теперь, когда вы знаете, для чего может использоваться MAC, перейдем к связанным с ним подвохам.

Листинг 3.2. Получение аутентифицированного сообщения в Rust

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn receive_message(key: &[u8], message: &[u8],
    authentication_tag: &[u8]) -> bool {
        let mut mac = Hmac::<Sha256>::new(key);
        mac.update(message);
        mac.verify(&authentication_tag).is_ok()
}
```

Получателю нужно воссоздать аутентификационный тег из того же ключа и сообщения

Проверяет, совпадает ли воспроизведенный тег с полученным

3.3. СВОЙСТВА БЕЗОПАСНОСТИ ИМИТОВСТАВКИ

Имитовставки, как и все криптографические примитивы, имеют свои особенности и подводные камни. Прежде чем продолжить, я дам несколько пояснений, какие свойства безопасности обеспечивают MAC и как их правильно применять. Вы узнаете (именно в таком порядке), что:

- имитовставки устойчивы к подделке аутентификационных тегов;
- аутентификационный тег считается безопасным, если имеет минимальную длину;
- примитивно аутентифицированные сообщения могут быть воспроизведены;
- проверка аутентификационного тега склонна к ошибкам.

3.3.1. Подделка аутентификационного тега

Главная задача MAC, связанная с безопасностью, состоит в том, чтобы предотвратить *подделку аутентификационного тега* нового сообщения. Не зная секретного ключа k, нельзя вычислить аутентификационный тег $t = \text{MAC}(k, m)$ для выбранных сообщений m. Звучит логично, не так ли? Мы не можем вычислить функцию, если нам не хватает какого-либо аргумента.

Однако MAC обеспечивают гораздо более надежную защиту. Реальные приложения часто позволяют злоумышленникам получать аутентификационные теги на некоторые сообщения. Например, так было в нашем сценарии во введении к этой главе, где пользователь мог получить практически произвольный аутентификационный тег, зарегистрировавшись под доступным псевдонимом. Следовательно, MAC должны быть защищены даже от таких продвинутых злоумышленников. Как правило, вместе с MAC идет доказательство того, что, даже если злоумышленник попросит вас создать аутентификационные теги для большого числа произвольно выбранных сообщений, он не сможет самостоятельно подделать аутентификационный тег для нового сообщения.

ПРИМЕЧАНИЕ

Можно задаться вопросом: насколько полезно доказательство такого экстремального свойства? Если злоумышленник может напрямую запрашивать аутентификационные теги на произвольно выбранные сообщения, то что тогда остается защищать? Но именно так и работают доказательства безопасности в криптографии: берется самый продвинутый злоумышленник и показывается, что даже его попытки навредить тщетны. На практике, как правило, взломщик не так силен. Таким образом, есть гарантия, что если даже сильный злоумышленник не может сделать что-то плохое, то у более слабого еще меньше возможностей.

По сути, защита от таких подделок работает *до тех пор, пока секретный ключ, используемый вместе с MAC, остается секретным*. Из этого следует, что секретный ключ должен быть достаточно случайным (подробнее об этом — в главе 8) и достаточно большим (обычно 16 байт). Кроме того, имитовставка уязвима для уже упомянутого в главе 2 типа неоднозначной атаки. Если вам нужно аутентифицировать структуры, перед этим обязательно сериализуйте их с помощью MAC. В противном случае вполне вероятно, что подделка станет тривиальной задачей.

3.3.2. Длина аутентификационного тега

Другой возможной атакой на MAC являются *коллизии*. Напомню, что найти коллизию для хеш-функции означает найти два различных входных блока А и Б, таких, что хеш(А) = хеш(Б). Это определение верно и для имитовставок: коллизия считается найденной при MAC(k, А) = MAC(k, Б) для входных блоков А и Б.

Как мы выяснили в главе 2 на примере ограниченности дней рождения, вероятность нахождения коллизии высока при малой длине выхода алгоритма. Например, в случае с имитовставкой злоумышленник, имеющий доступ к сервису, производящему 64-разрядные аутентификационные теги, может с высокой вероятностью найти коллизию, запросив гораздо меньшее число тегов (2^{32}). Такие атаки практикуют редко, однако существуют сценарии, в которых устойчивость к коллизиям имеет значение. По этой причине необходим такой размер аутентификационного тега, который ограничивал бы подобные атаки. Обычно используются 128-битные аутентификационные теги, так как они обеспечивают достаточную устойчивость.

[Запрос 2^{64} аутентификационных тегов] займет 250 000 лет в непрерывном канале связи со скоростью 1 Гбит/с, и только если в течение всего этого времени секретный ключ K останется прежним.

RFC 2104 («HMAC: Имитовставка, использующая хеш-функции с ключом», 1997)

Применение 128-битного аутентификационного тега может показаться нелогичным, поскольку выход хеш-функции должен быть длиной 256 бит. Однако

хеш-функции являются публичными алгоритмами, которые можно вычислить в *автономном режиме*, что позволяет злоумышленнику оптимизировать и распараллелить атаку. Такая функция с ключом, как MAC, вынуждает его напрямую запрашивать аутентификационные теги, что обычно существенно замедляет атаку. Таким образом, если длина аутентификационного тега 128 бит, то для того, чтобы найти коллизию с вероятностью 50 %, злоумышленник должен сделать 2^{64} онлайн-запроса. Это уже достаточно большой показатель. Тем не менее при желании можно увеличить аутентификационный тег до 256 бит.

3.3.3. Атаки повторного воспроизведения

Я еще не упомянул об *атаках повторного воспроизведения*. Представьте, что Алиса и Боб открыто общаются, используя незащищенное соединение. Они защищают свои сообщения от подделки, добавляя аутентификационный тег к каждому из них. Более того, в соответствии с передовой практикой оба они имеют собственный секретный ключ для защиты своей стороны соединения. Процесс их общения изображен на рис. 3.5.



Рис. 3.5. Два пользователя, имеющие два ключа, k1 и k2, обмениваются сообщениями, содержащими аутентификационные теги. Эти теги вычисляются из k1 или k2 в зависимости от направления сообщения. Злоумышленник пересылает одно из сообщений пользователю

В этом случае ничто не мешает злоумышленнику воспроизвести одно из сообщений для его получателя. Протокол, в основе которого лежит имитовставка, должен знать об этом и иметь средства защиты от такого вмешательства. Один из способов — добавление счетчика прямого отсчета на вход MAC (рис. 3.6).

Как правило, счетчики имеют фиксированную длину 64 бита. Такой размер позволяет отправить 2^{64} сообщения до достижения лимита счетчика (с риском его обхода и повторения).



Рис. 3.6. Два пользователя, имеющие два ключа, k1 и k2, обмениваются сообщениями, содержащими аутентификационные теги. Эти теги вычисляются из k1 или k2 в зависимости от направления сообщений. Злоумышленник пересылает одно из сообщений пользователю. Поскольку жертва увеличила свой счетчик, тег, вычисленный на основании «2, хорошо, а у тебя?», не совпадет с тегом злоумышленника, что позволяет жертве успешно заблокировать воспроизведенное сообщение

Конечно, если общий секрет часто меняется (то есть после какого-то числа сообщений участники соглашаются использовать новый общий секрет), то показатель может быть уменьшен и сброшен до 0 после каждой смены ключа. (Вы должны убедить себя в том, что повторное применение одного и того же счетчика с двумя разными ключами — это нормально.) Опять же счетчики *не могут иметь переменную длину* из-за неоднозначных атак.

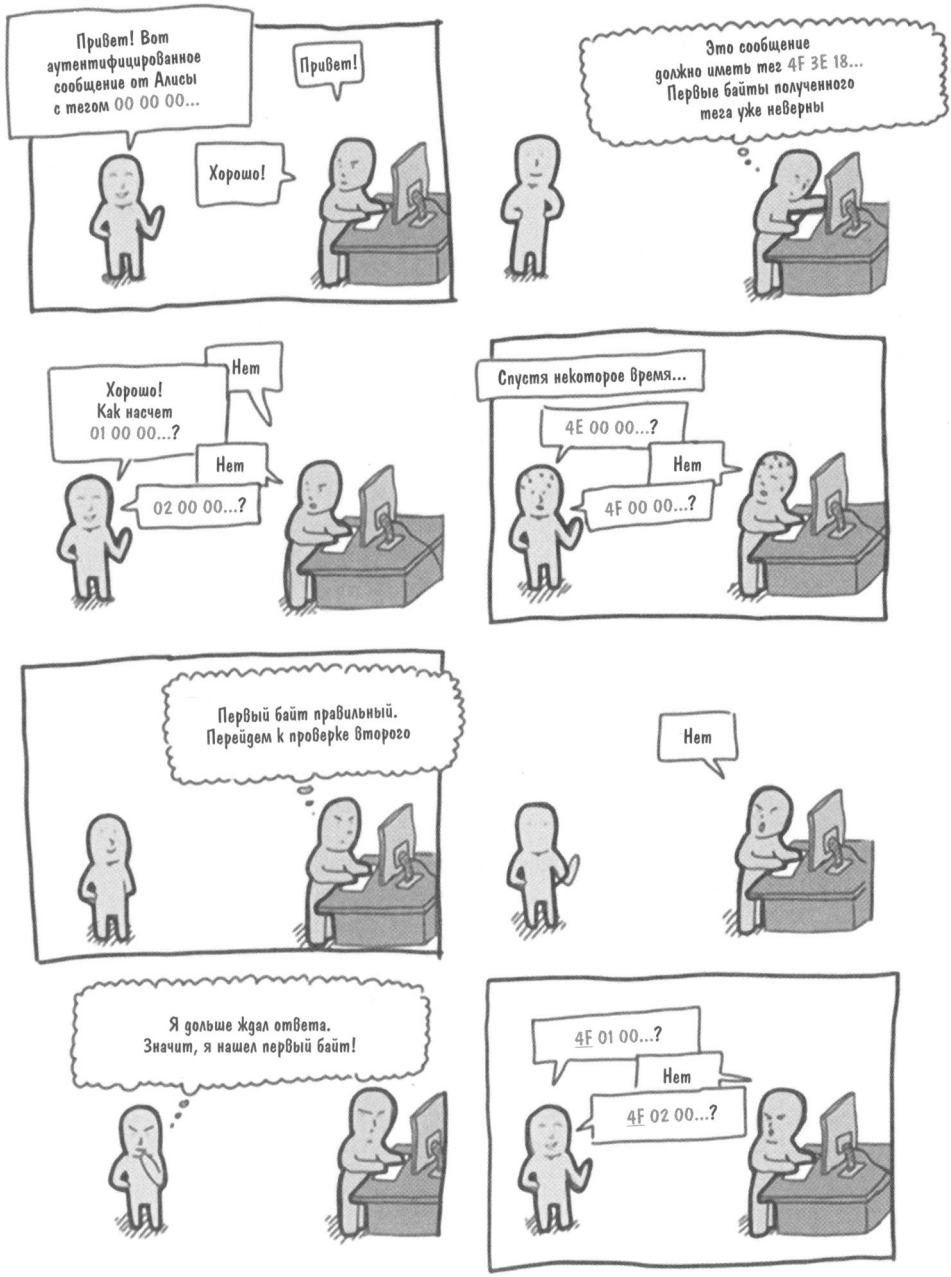
УПРАЖНЕНИЕ

Как вы думаете, каким образом счетчик переменной длины мог бы позволить злоумышленнику подделать аутентификационный тег?

3.3.4. Проверка аутентификационных тегов за постоянное время

Последняя проблема очень важна для меня, так как во время аудита я неоднократно находил эту уязвимость в приложениях. При проверке аутентификационного тега сравнение полученного тега с вычисленным должно занимать *постоянное время*. Это означает, что сравнение всегда должно занимать одно и то же время при условии, что полученный тег имеет правильный размер. Если время сравнения двух аутентификационных тегов варьируется, значит, скорее всего, алгоритм возвращается в момент, когда сравниваемые теги различаются. Обычно это дает

злоумышленнику достаточно информации для проведения атак, которые позволяют воссоздать актуальный аутентификационный тег байт за байтом, измеряя время, необходимое для завершения проверки. В следующем комиксе объяснена эта проблема. Мы называем такие атаки *атаками по времени*.



К счастью, криптографические библиотеки, реализующие MAC, также предоставляют удобные функции для проверки аутентификационного тега за постоянное время. Если вам интересно, что они собой представляют, то в листинге 3.3 показано, как Golang реализует сравнение аутентификационных тегов в коде постоянного времени.

Листинг 3.3. Выполнение сравнения за постоянное время в Golang

```
for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
}
```

Фокус заключается в том, что здесь пропускается любое ветвление. Ответ на вопрос, как именно это работает, мы оставляем читателю в качестве задания.

3.4. ИМИТОВСТАВКА В РЕАЛЬНОМ МИРЕ

Теперь, когда я рассказал о том, что такое имитовставки и какие свойства безопасности они обеспечивают, посмотрим, как их применяют в реальных условиях. Этой теме посвящены следующие разделы.

3.4.1. Аутентификация сообщения

Имитовставки применяются во многих местах, чтобы гарантировать подлинность коммуникации между двумя машинами или двумя пользователями. Она необходима и когда сообщения передаются открытым текстом, и когда они зашифрованы. Я уже объяснял, как обеспечивается подлинность сообщения, передающегося открытым текстом, а в следующей главе расскажу, как это происходит при шифровании сообщения.

3.4.2. Получение ключей

Одна из особенностей имитовставок состоит в том, что часто они предназначены для генерации байтов случайного вида, подобно хеш-функциям. Это свойство можно использовать, чтобы реализовать один ключ для генерации случайных чисел или создать несколько ключей. В главе 8, посвященной секретам и случайности, я представлю функцию получения ключа на основе HMAC (HKDF) — одного из алгоритмов MAC, о котором поговорим чуть позже.

ПСЕВДОСЛУЧАЙНАЯ ФУНКЦИЯ

Представьте себе набор из всех функций, которые принимают входные данные переменной длины и возвращают случайный результат фиксированного размера. Было бы замечательно, если бы мы могли выбрать из него произвольную функцию и использовать ее в качестве имитовставки (без ключа). Оставалось бы договориться о конкретной функции, как при согласовании ключа. К сожалению, у нас нет такой возможности, поскольку набор слишком велик. Тем не менее мы можем эмулировать выбор случайной функции, разработав нечто

довольно близкое к ней — *псевдослучайную функцию* (pseudorandom function, PRF). HMAC и большинство практических MAC — именно такие конструкции. Они рандомизируются по аргументу ключа. Выбор другого ключа аналогичен выбору случайной функции.

УПРАЖНЕНИЕ

Внимание: не все MAC являются PRF. Как вы думаете — почему?

3.4.3. Целостность файлов cookie

Для отслеживания браузерных сессий пользователей можно посылать им случайную строку, связанную с их метаданными, или сами метаданные с вложенным аутентификационным тегом, чтобы они не могли их изменить. Это то, о чем мы говорили в примере, представленном во вводной части главы.

3.4.4. Хеш-таблицы

Языки программирования обычно предоставляют такие структуры данных, как *хеш-таблицы* (также называемые хеш-картами, словарями, ассоциированными массивами и т. д.), которые используют некриптографические хеш-функции. Если сервис предоставляет хеш-таблицы таким образом, что входные данные некриптографической хеш-функции могут контролироваться злоумышленниками, против него может быть произведена атака *«отказ в обслуживании»* (denial of service attacks, DoS), которая подразумевает выведение сервиса из строя. Для предотвращения такой атаки некриптографическая хеш-функция обычно рандомизируется в начале работы программы.

Во многих крупных приложениях вместо некриптографической хеш-функции используется имитовставка со случайным ключом. Так обстоит дело во многих языках программирования, например Rust, Python, Ruby, или в крупных приложениях, например в ядре Linux. Все они применяют так называемый *SipHash*, то есть MAC, оптимизированный для получения коротких аутентификационных тегов со случайным ключом, который генерируется в начале работы программы.

3.5. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ ИМИТОВСТАВОК

Итак, имитовставки представляют собой криптографические алгоритмы, которые могут применяться между одной или несколькими сторонами для сохранения целостности и подлинности информации. Поскольку широко распространенные имитовставки являются также хорошими источниками случайности, их часто задействуют для детерминированной генерации случайных чисел в различных типах

алгоритмов, например в алгоритме одноразовых паролей с временным параметром (time-based one-time password, TOTP), с которым вы познакомитесь в главе 11. В этом разделе рассмотрим два стандартных алгоритма имитозащиты, которые можно задействовать в настоящее время, — HMAC и KMAC.

3.5.1. HMAC — имитовставка с использованием хеш-функции

Наиболее распространенным MAC является *HMAC* (hash-based MAC), изобретенный в 1996 году М. Беллар, Р. Канетти и Х. Кравчиком и описанный в RFC 2104, FIPS Publication 198 и ANSI X9.71. Как следует из названия, HMAC представляет собой способ применения хеш-функций с ключом. Использование хеш-функции для построения MAC — популярная концепция, поскольку хеш-функции широко распространены, быстро реализуются с помощью ПО и имеют аппаратную поддержку в большинстве систем. Вспомните, в главе 2 я говорил о том, что SHA-2 не следует использовать непосредственно для хеширования секретов из-за ее уязвимости к *атакам удлинением сообщения* (подробнее об этом — в конце главы). Как же определить, каким образом преобразовать хеш-функцию в функцию с ключом? Эту задачу решает HMAC. С точки зрения внутреннего устройства HMAC выполняет следующие шаги, наглядно продемонстрированные на рис. 3.7.

1. Сначала из основного ключа получают два ключа, $k1 = k \oplus ipad$ и $k2 = k \oplus opad$, где *ipad* (внутреннее дополнение, от inner padding) и *opad* (внешнее дополнение, от outer padding) — константы, $\oplus$ — символ операции XOR.
2. Затем HMAC объединяет ключ $k1$ с сообщением и хеширует его.
3. Полученный результат объединяется с ключом $k2$ и хешируется еще раз.
4. Таким образом получается конечный аутентификационный тег.

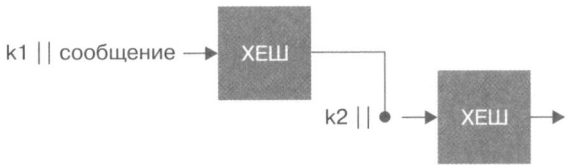


Рис. 3.7. HMAC работает путем хеширования конкатенации (||) ключа k1 и входного сообщения с последующим хешированием конкатенации ключа k2 с результатом предыдущей операции; k1 и k2 детерминированно выводятся из секретного ключа k

Поскольку HMAC — это настраиваемый алгоритм, размер его аутентификационного тега определяется применяемой хеш-функцией. Например, HMAC-SHA-256 использует SHA-256 и выдает аутентификационный тег размером 256 бит, HMAC-SHA-512 выдает аутентификационный тег размером 512 бит и т. д.

ВНИМАНИЕ

Как мы говорили ранее, несмотря на то что размер выхода HMAC можно усечь, аутентификационный тег должен быть не менее 128 бит. Тем не менее это правило соблюдается не всегда, так как некоторые приложения обходятся 64 битами из-за явной необходимости обработки ограниченного количества запросов. Такой подход предусматривает некоторые компромиссы, и опять же, прежде чем делать что-то нестандартное, важно изучить все мелочи.

Конструкция HMAC обусловлена необходимостью упрощения доказывания. В ряде работ подтверждена надежность защиты HMAC от подделок при условии, что хеш-функция в его основе обладает свойствами, которыми должны обладать все безопасные хеш-функции с криптографической точки зрения. Благодаря этому мы можем использовать HMAC в сочетании с большим количеством хеш-функций. Сегодня HMAC в основном применяется с SHA-2.

3.5.2. KMAC — имитовставка с использованием cSHAKE

Поскольку SHA-3 неуязвима для атак удлинением сообщения (что было одним из условий конкурса по разработке SHA-3), не имеет смысла задействовать этот алгоритм с HMAC вместо, например, эффективного `SHA-3-256(ключ || сообщение)`, чем по сути и является *KMAC*.

KMAC использует cSHAKE — настраиваемую версию функции расширенного выхода SHAKE (XOF), рассмотренную в главе 2. Он однозначно шифрует MAC-ключ, входное сообщение и требуемую длину выхода (KMAC является своего рода расширенным выходом MAC) и отдает это все cSHAKE в качестве входных данных для впитывания (рис. 3.8). KMAC также использует «KMAC» в качестве имени функции (для настройки cSHAKE) и, ко всему прочему, может принимать заданную пользователем строку настройки.

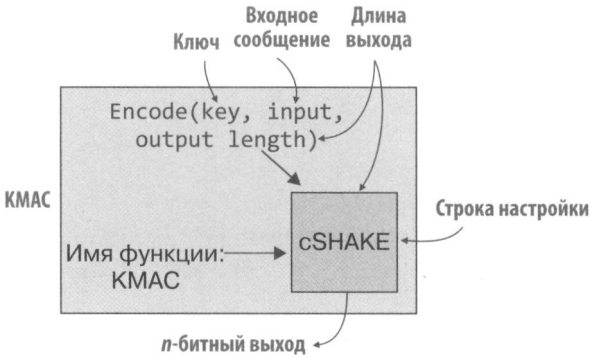


Рис. 3.8. KMAC — это просто обертка вокруг cSHAKE. Чтобы применять ключ, он шифрует (однозначным образом) ключ, входное сообщение и длину выхода и передает результат на вход cSHAKE

Интересно отметить: поскольку KMAC впитывает требуемую длину выхода, то несколько запросов с разной длиной выхода будут иметь совершенно разные результаты, что, как правило, нетипично для XOF. Эта особенность делает KMAC довольно универсальной функцией.

3.6. SHA-2 И АТАКИ УДЛИНЕНИЕМ СООБЩЕНИЯ

Мы уже неоднократно говорили о том, что не следует хешировать секреты с помощью SHA-2, поскольку она неустойчива к *атакам удлинением сообщения*. В этом разделе попытаемся дать простое объяснение этой атаки.

Вернемся к примеру из вводной части главы — к тому шагу, когда мы попытались применить SHA-2 для защиты целостности файла cookie. Если вы помните, этот вариант оказался неприемлемым, так как пользователь может подделать cookie (например, добавив поле `admin=true`) и пересчитать его хеш. Поскольку SHA-2 — публичная функция, ничто не помешает ему сделать это. На рис. 3.9 изображено такое вмешательство.



Рис. 3.9. Веб-страница посылает файл cookie и его хеш пользователю. Для аутентификации последний должен отправлять этот cookie при каждом новом запросе. К сожалению, злоумышленник может подделать cookie и пересчитать его хеш, нарушив тем самым проверку целостности. В результате веб-страница принимает новый файл cookie, как будто он верный

Если добавить секретный ключ к хешируемому элементу, пользователь не сможет пересчитать дайджест, поскольку для этого ему потребуется секретный ключ, как в случае с MAC. Таким образом, при получении поддельного файла cookie страница вычисляет алгоритм `SHA-256(ключ || поддельный_cookie)`, где `||` представляет

собой конкатенацию двух значений, и получает результат, не совпадающий с тем, что, вероятно, прислал злоумышленник. Этот подход иллюстрирует рис. 3.10.



Рис. 3.10. Использование ключа при вычислении хеша cookie позволяет предположить, что злоумышленник, желающий подделать собственный cookie, не сможет вычислить правильный дайджест по новому cookie. Однако, как мы увидим далее, для SHA-256 этот вариант не работает

К сожалению, SHA-2 имеет досадную особенность: из дайджеста по входным данным можно вычислить дайджест входного сообщения и даже больше. Что это означает? Давайте посмотрим на рис. 3.11, где показано использование SHA-256 в виде `SHA-256(секрет || входные_данные1)`.

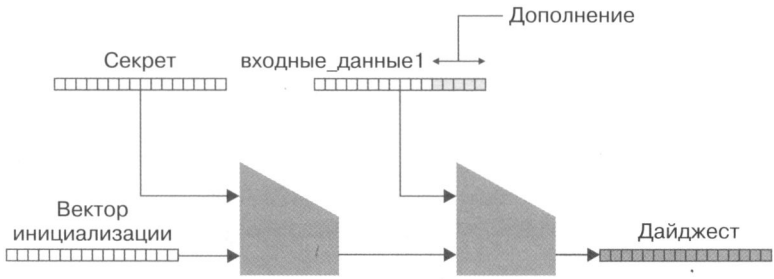


Рис. 3.11. SHA-256 хеширует секрет, конкатенированный с файлом cookie (входные_данные1). Напомню, что SHA-256 использует структуру Меркла — Дамгарда для итеративного вызова функции сжатия над блоками входных данных, начиная с вектора инициализации (initialization vector, IV)

Этот рисунок сильно упрощен, но представьте, что `входными_данными1` является строка `user=боб`. Обратите внимание: полученный дайджест — это фактически

полное промежуточное состояние хеш-функции на данном этапе. Ничто не мешает сделать вид, что дополнение является частью входного блока, и продолжить круговорот Меркла – Дамгарда. Эта атака проиллюстрирована на рис. 3.12: с помощью дайджеста вычисляется хеш `входных_данных1 || дополнения || входных_данных2`. В нашем примере `входные_данные2` – это `&admin=true`.

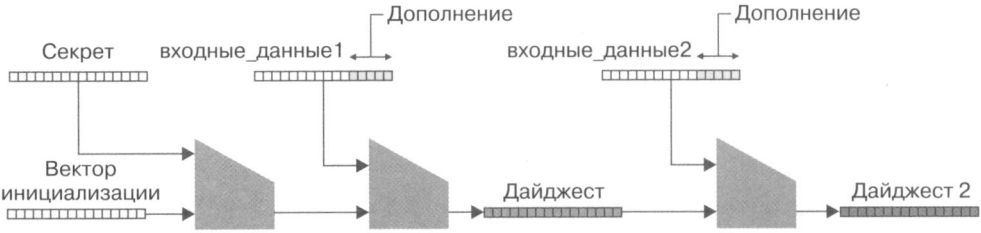


Рис. 3.12. Выход, который представляет собой хеш SHA-256 cookie (средний дайджест), используется для расширения хеша на большее количество данных. В результате создается хеш (правый дайджест) секрета, конкатенированного с входными_данными1, первыми байтами дополнения и входными_данными2

Эта уязвимость позволяет продолжить хеширование от заданного дайджеста, как будто операция не была завершена, что нарушает предыдущий протокол (рис. 3.13).



Рис. 3.13. Злоумышленник успешно использует атаку удлинением сообщения для подделки cookie и вычисляет правильный хеш с помощью предыдущего хеша

Тот факт, что первое дополнение теперь должно быть частью входных данных, может предотвратить использование некоторых протоколов. Тем не менее самое незначительное изменение может вновь привести к появлению уязвимости. По этой причине *никогда* не следует хешировать секреты с помощью SHA-2. Безусловно, существует несколько способов корректного хеширования секретов с поддержкой HMAC (например, рабочим вариантом является `SHA-256(k || сообщение || k)`). Таким образом, если вы хотите использовать SHA-2, возьмите HMAC, а если SHA-3 – KMAC.

РЕЗЮМЕ

- Имитовставки — это симметричные криптографические алгоритмы, позволяющие одной или нескольким сторонам, имеющим один и тот же ключ, подтвердить целостность и подлинность сообщений.
  - Для проверки подлинности сообщения и связанного с ним аутентификационного тега можно пересчитать аутентификационный тег сообщения и секретный ключ, а затем сверить полученный результат и исходный тег. Если они различаются, значит, сообщение было подделано.
  - Сравнение полученного аутентификационного тега с вычисленным должно проходить за постоянное время.
- Хотя имитовставки по умолчанию защищают целостность сообщений, они не распознают воспроизведенные сообщения.
- Стандартизированными и хорошо зарекомендовавшими себя MAC являются HMAC и KMAC.
- HMAC можно задействовать вместе с различными хеш-функциями. На практике он часто используется с SHA-2.
- Для предотвращения коллизий и подделок аутентификационные теги должны иметь минимальную длину 128 бит.
- Никогда не применяйте SHA-256 непосредственно для построения MAC, так как существует риск неправильной реализации. Всегда берите для этой цели функцию типа HMAC.

1 Аутентифицированное шифрование

https://t.me/it_boooks/2

В этой главе

- ✓ Симметричное и аутентифицированное шифрование.
- ✓ Широко используемые алгоритмы аутентифицированного шифрования.
- ✓ Другие виды симметричного шифрования.

Конфиденциальность — это защита данных от посторонних глаз, а шифрование — инструмент для достижения данной цели. Изначально именно ради шифрования и была создана криптография. Большинство первых криптографов задавались вопросом: «Как помешать сторонним наблюдателям понимать нас?» Если сначала эта наука развивалась и совершенствовалась в закрытых кругах, что приносило пользу только правительствам и военным, то теперь криптография доступна всему миру. Сегодня шифрование используется повсеместно для обеспечения конфиденциальности и безопасности в различных сферах нашей жизни. В этой главе мы рассмотрим, что такое шифрование, какие проблемы оно решает и как современные приложения активно задействуют этот криптографический примитив.

ПРИМЕЧАНИЕ

Чтобы лучше понять эту главу, рекомендую прочитать главу 3, посвященную имитовставкам.

4.1. ЧТО ТАКОЕ ШИФР

> Это что-то вроде сленга в разговоре с братом или сестрой о том, что вы будете делать после школы, чтобы мама не поняла, что вы задумали.
>
> *Натаниэль Л. (2020, https://twitter.com/Natanael_L)*

Предположим, Алиса и Боб хотят обменяться сообщениями. С практической точки зрения в их распоряжении имеется множество средств связи (почта, телефон, Интернет и т. д.), но каждое из них по умолчанию небезопасно. Почтальон может вскрыть их письма, операторы способны отслеживать звонки и текстовые сообщения, а интернет-провайдеры или любые сетевые серверы, находящиеся между Алисой и Бобом, могут получить доступ к содержимому пакетов, которыми они обмениваются.

Без лишних церемоний представим спасителя Алисы и Боба в этой ситуации: *алгоритм шифрования*, также называемый просто *шифром*. Пока будем рассматривать его как черный ящик, с помощью которого Алиса может шифровать свои сообщения Бобу. *Зашифровывая* сообщение, Алиса превращает его в нечто случайное. Для этого потребуется:

- *секретный ключ* — очень важно, чтобы этот элемент имел непредсказуемый, случайный и хорошо защищенный характер, поскольку криптостойкость (то есть безопасность) алгоритма шифрования напрямую зависит от секретности ключа. Подробнее расскажу об этом в главе 8, посвященной секретам и случайности;
- *открытый текст* — то, что нужно зашифровать. Это может быть текст, изображение, видео — все, что можно перевести в биты.

Результатом действия алгоритма шифрования является *шифротекст*, который и представляет собой зашифрованное содержимое. Теперь Алиса может спокойно использовать одно из перечисленных ранее средств связи для передачи его Бобу. Для тех, кто не знает секретного ключа, шифротекст будет выглядеть случайным набором символов, и никакая информация о содержании сообщения (открытом тексте) не будет раскрыта. Получив шифротекст, Боб может применить алгоритм расшифрования для его преобразования в исходный открытый текст. Для расшифрования потребуется:

- *секретный ключ* — тот же самый секретный ключ, который использовала Алиса для создания шифротекста. Поскольку для обоих алгоритмов нужен один и тот же ключ, его называют *симметричным ключом*. Именно поэтому мы также указываем, что задействуем *симметричное шифрование*, а не просто *шифрование*;
- *шифротекст* — зашифрованное сообщение, которое Боб получил от Алисы.

Таким образом раскрывается исходный открытый текст. Процессы шифрования и расшифрования изображены на рис. 4.1.

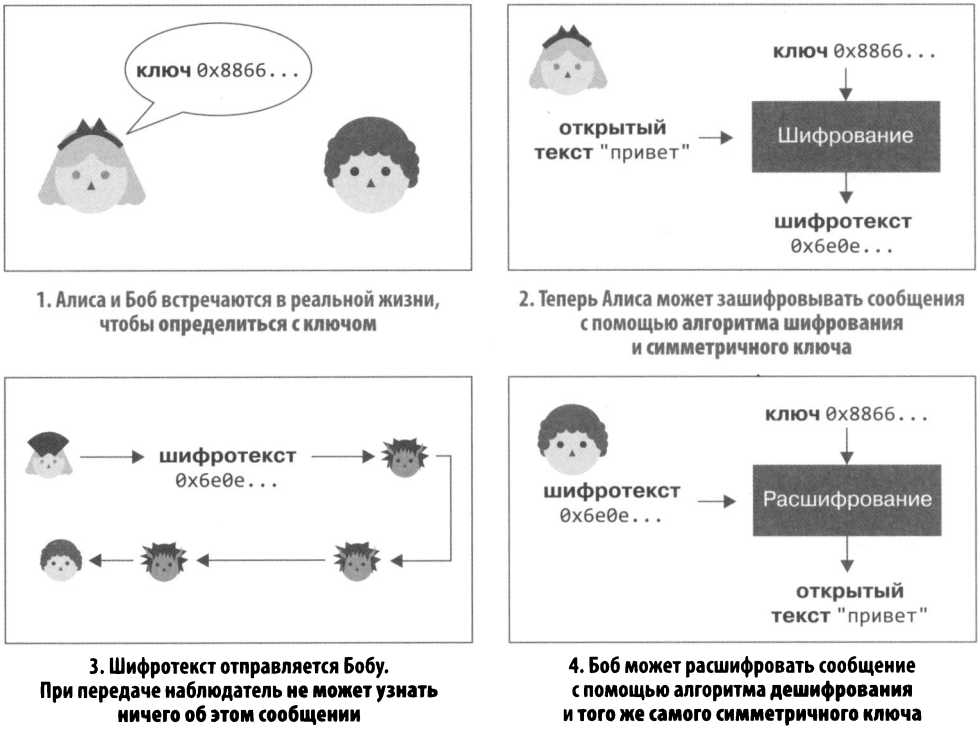


Рис. 4.1. Алиса *(вверху справа)* шифрует открытый текст «привет» ключом 0x8866... (сокращенное шестнадцатеричное число). Затем она отправляет зашифрованный текст Бобу. Боб *(справа внизу)* расшифровывает полученный шифротекст, используя тот же ключ и алгоритм расшифрования

Шифрование позволяет Алисе преобразовать свое сообщение в нечто, что выглядит случайным образом и может быть безопасно передано Бобу. Расшифрование позволяет ему вернуть зашифрованное сообщение к его исходному виду. Этот криптографический примитив обеспечивает конфиденциальность (или секретность, или приватность) их сообщений.

ПРИМЕЧАНИЕ

Каким образом Алиса и Боб определились, какой симметричный ключ они будут применять? Пока предположим, что один из них воспользовался алгоритмом генерации непредсказуемых ключей и при личной встрече передал сгенерированный ключ другому. На практике запуск подобных протоколов с общими секретами часто становится серьезной проблемой для компаний. В этой книге вы увидите множество решений данной задачи.

Обратите внимание: я еще ничего не сказал о том, что вынесено в название этой главы, — об аутентифицированном шифровании (authenticated encryption).

До этого я говорил просто о шифровании, которое само по себе небезопасно (об этом подробнее — позже). Однако необходимо объяснить принцип его работы, прежде чем представить аутентифицированное шифрование. Так что наберитесь терпения, поскольку я сначала расскажу об основном стандарте шифрования — *Advanced Encryption Standard* (AES).

4.2. БЛОЧНЫЙ ШИФР ADVANCED ENCRYPTION STANDARD

В 1997 году NIST объявил открытый конкурс на разработку *Advanced Encryption Standard*, который должен был заменить уже начавший устаревать алгоритм Data Encryption Standard (DES). Конкурс длился три года, и за это время команды криптографов из разных стран представили 15 проектов. По итогам конкурса победителем был признан один — Rijndael Винсента Риджмена и Джоан Демен.

В 2001 году NIST выпустил AES в публикации FIPS (Federal Information Processing Standards) 197. Данный алгоритм до сих пор является основным используемым шифром. В этом разделе я объясню, как он работает.

4.2.1. Какой уровень криптостойкости обеспечивает AES

AES имеет три версии: в AES-128 применяется ключ длиной 128 бит (16 байт), в AES-192 — длиной 192 бита (24 байта) и в AES-256 — длиной 256 бит (32 байта). Длина ключа определяет уровень криптостойкости — *чем больше, тем лучше*. Тем не менее в большинстве приложений используется AES-128, поскольку он обеспечивает достаточную защиту (128 бит защиты).

Термином «битная криптостойкость» обычно обозначается уровень безопасности криптографических алгоритмов. Например, в алгоритме AES-128 указано, что для проведения самой опасной атаки потребуется около 2^{128} операций. Это гигантское число, и именно к такому уровню криптостойкости стремится большинство приложений.

БИТНАЯ КРИПТОСТОЙКОСТЬ КАК ВЕРХНИЙ ПРЕДЕЛ

Тот факт, что 128-битный ключ обеспечивает 128 бит защиты, характерен именно для AES. Это значение не является золотым стандартом. Теоретически 128-битный ключ в каком-нибудь другом алгоритме может обеспечить защиту менее 128 бит, но никогда — больше этого значения (всегда есть возможность атаки методом грубой силы). Перебор всех возможных ключей потребует не более 2^{128} операций, что означает снижение криптографической стойкости минимум до 128 бит.

Насколько велико число 2^{128}? Здесь нужно помнить, что каждая следующая степень числа 2 удваивает конечный результат. Например, 2^3 в два раза больше, чем 2^2. Если 2^{100} операций практически невозможно провести, то представьте, что будет, если таких операций будет вдвое больше (2^{101}). Чтобы получить 2^{128}, нужно удвоить основание в 128 раз! Проще говоря, 2^{128} — это 340 недециллионов 282 дециллиона 366 нониллионов 920 октиллионов 938 септиллионов 463 секстиллиона 463 квинтиллиона 374 квадриллиона 607 триллионов 431 миллиард 768 миллионов 211 тысяч 456. Трудно представить, насколько велико это число, но можно предположить, что на практике мы никогда не сможем его достичь. К тому же мы не учли столь же огромный объем пространства, необходимый для работы большой и сложной атаки.

Очевидно, AES-128 будет оставаться надежным алгоритмом еще долго. По крайней мере, пока в криптоанализе не будет найдена новая уязвимость, позволяющая сократить количество операций, необходимых для атаки на алгоритм.

4.2.2. Интерфейс AES

В интерфейсе шифрования AES мы видим следующее.

- Как уже говорилось, алгоритм принимает ключ переменной длины.
- Он также принимает открытый текст длиной ровно 128 бит.
- На выходе выдает шифротекст длиной ровно 128 бит.

Поскольку AES шифрует открытый текст фиксированного размера, мы называем его *блочным шифром*. Некоторые другие шифры, представленные далее в этой главе, могут преобразовывать открытые тексты произвольной длины.

Операция расшифрования в точности обратна: берется тот же ключ и шифротекст длиной 128 бит и возвращается исходный 128-битный открытый текст. По сути, расшифрование — это процесс, обратный шифрованию. Оно возможно благодаря тому, что операции шифрования и расшифрования *детерминированы*, то есть дают одинаковый результат независимо от количества запросов.

С технической точки зрения блочный шифр с ключом представляет собой *перестановку*: он сопоставляет все возможные открытые тексты со всеми возможными шифротекстами (рис. 4.2). Использование другого ключа меняет результат сопоставления. Перестановка также является обратимой. Из шифротекста можно получить соответствующий ему открытый текст (иначе расшифровка была бы невозможна).

Конечно, не хватит места, чтобы привести список всех возможных открытых текстов и соответствующих им шифротекстов; 128-битный блочный шифр подразумевает 2^{128} сопоставлений. Вместо этого мы разрабатываем конструкции типа AES, которые работают как перестановки и рандомизируются ключом. Мы говорим о них как о *псевдослучайных перестановках* (pseudorandom permutations, PRPs).

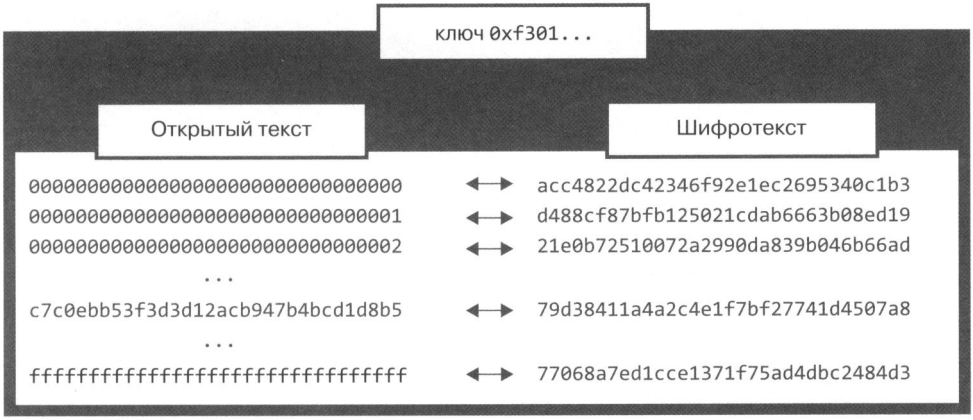


Рис. 4.2. Шифр с ключом можно рассматривать как перестановку: он сопоставляет все возможные открытые тексты со всеми возможными шифротекстами

4.2.3. Внутренняя структура AES

Немного углубимся в суть AES и посмотрим, что происходит внутри него. Отмечу, что AES рассматривает *состояние* открытого текста в процессе шифрования как матрицу байтов 4 × 4 (рис. 4.3).

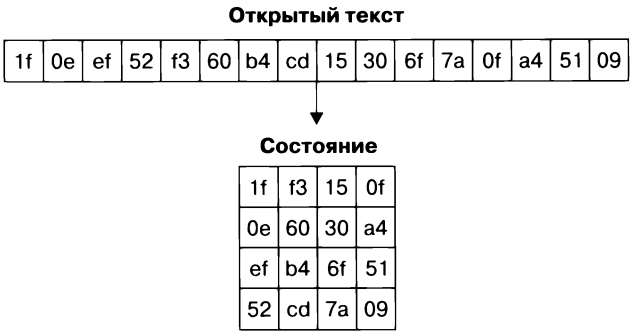


Рис. 4.3. На входе в алгоритм AES открытый текст размером 16 байт преобразуется в матрицу 4 × 4. Затем это состояние шифруется и преобразуется в 16-байтовый шифротекст

С практической точки зрения это не имеет особого значения, но именно таково определение AES. По принципу работы AES напоминает многие аналогичные симметричные криптографические примитивы, называемые *блочными шифрами*, то есть шифрами, которые шифруют блоки фиксированного размера. AES также имеет *раундовую функцию*, которую повторяет несколько раз, начиная с исходного блока входных данных (открытого текста) (рис. 4.4).

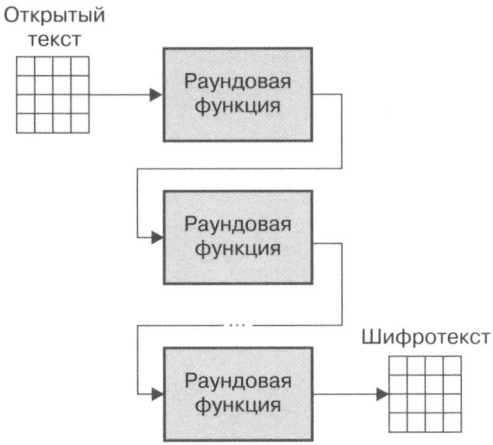


Рис. 4.4. AES шифрует состояние, многократно выполняя над ним раундовую функцию. Эта функция принимает несколько аргументов, включая секретный ключ (для упрощения схемы на рисунке их нет)

При каждом запросе раундовой функции происходит дополнительное преобразование состояния, в результате чего получается шифротекст. В каждом раунде используется свой *раундовый ключ*, который получается из основного симметричного ключа в ходе так называемого *развертывания ключа*. Таким образом, при малейшем изменении битов симметричного ключа выполняется совершенно другое шифрование (этот принцип называется *диффузией*).

Раундовая функция состоит из множества операций, которые перемешивают и меняют байты состояния. В AES она использует четыре подфункции. Я не стану рассказывать, как именно они работают (эту информацию можно найти в любой книге по AES), но назову их: `SubBytes` (замена байтов), `ShiftRows` (сдвиг строк), `MixColumns` (смешивание столбцов) и `AddRoundKey` (добавление раундового ключа). Первые три операции легко обратимы (по их результату можно найти входные значения), а вот последняя — нет. Она выполняет операцию исключающего ИЛИ (XOR) с раундовым ключом и состоянием и, таким образом, требует знания раундового ключа для обратного преобразования. На рис. 4.5 показано, что входит в раунд.

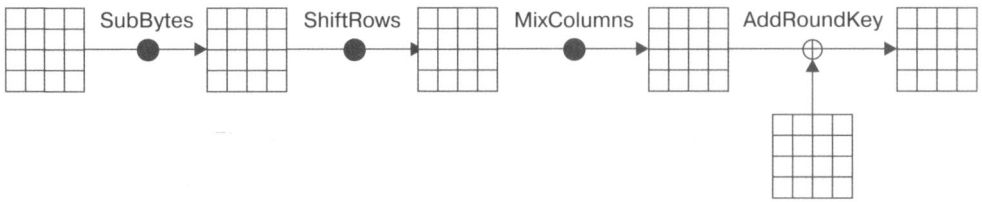


Рис. 4.5. Типичный раунд AES. (В первом и последнем раундах некоторые операции пропускаются.) Четыре различные функции преобразуют состояние. Каждую из них можно обратить, так как в противном случае расшифровка будет невозможна

Количество итераций раундовой функции в AES, которая на практике применяется на уменьшенном числе раундов, было выбрано таким образом, чтобы предотвратить криптоанализ. Например, существуют чрезвычайно эффективные методы *полного взлома* (атак, восстанавливающих ключ), направленные на трехраундовые варианты AES-128. В результате многократных итераций шифр преобразует открытый текст в нечто совершенно не похожее на исходный открытый текст. Малейшее изменение открытого текста даст совершенно другой шифротекст. Этот принцип называется *лавинным эффектом*.

ПРИМЕЧАНИЕ

Как правило, реальные криптографические алгоритмы сравниваются по криптостойкости, размеру и скорости. Мы уже поговорили о криптостойкости и размере AES: его криптостойкость зависит от размера ключа, и он может шифровать 128-битные блоки данных за раз. Что касается скорости, то многие производители процессоров реализовали AES в аппаратной части. Например, в процессорах Intel и AMD введен набор инструкций AES New Instructions (AES-NI), который может быть использован для эффективной реализации шифрования и расшифрования посредством AES. Эти специальные инструкции значительно увеличили реальную скорость работы AES.

Возможно, у вас остался вопрос: как с помощью AES зашифровать данные размером больше или меньше 128 бит? На него я отвечу далее.

4.3. ЗАШИФРОВАННЫЙ ПИНГВИН И РЕЖИМ СЦЕПЛЕНИЯ БЛОКОВ ШИФРОТЕКСТА

Теперь, когда мы познакомились с блочным шифром AES и немного с его внутренним устройством, посмотрим, как применять этот алгоритм на практике. Проблема с блочным шифром заключается в том, что он шифрует информацию только блоками. Чтобы зашифровать что-то размером не 128 бит, необходимо использовать *дополнение*, а также *режим шифрования*. Поэтому поговорим о том, что означают эти понятия.

Представьте, что вам нужно зашифровать длинное сообщение. Сначала можно просто разделить его на блоки по 16 байт (размер блока в AES). Затем, если бы последний блок открытого текста получился меньше 16 байт, вы могли бы заполнить его, добавив в конец несколько байтов. В этом и заключается суть дополнения!

Существует несколько вариантов выбора *дополняющих байтов*, но наиболее важный аспект дополнения — такое обязательное требование к нему, как обратимость. После расшифровки шифротекста мы должны иметь возможность удалить дополнение и получить исходное сообщение без него. Простое добавление случайных байтов не сработает, поскольку в таком случае невозможно определить, какие байты входят в исходное сообщение.

Наиболее популярным механизмом дополнения часто называют *PKCS#7*, который впервые появился в стандарте PKCS#7, опубликованном компанией RSA в конце 1990-х годов. Дополнение PKCS#7 устанавливает правило: каждый

дополняющий байт должен иметь значение длины требуемого дополнения. А что, если открытый текст уже состоит из 16 байт? Тогда добавляется целый блок дополнения, состоящий из байтов со значением 16. Я наглядно проиллюстрировал это правило на рис. 4.6. Таким образом, дополнение можно удалить, выполнив простую проверку значения последнего байта открытого текста, показывающего длину этого дополнения.



Рис. 4.6. Если открытый текст не достигает размера блока, то он дополняется байтами до необходимой длины. Здесь открытый текст имеет размер 8 байт, поэтому мы заполняем его еще 8 байтами (со значением 8), чтобы его длина стала равной 16 байтам, необходимым для шифрования AES

Теперь необходимо рассказать об одной большой проблеме. До сих пор для шифрования длинного сообщения достаточно было разделить его на блоки по 16 байт и, возможно, добавить последний блок. Этот нехитрый прием называется *режимом электронной кодовой книги* (Electronic Codebook, ECB). Как вы уже знаете, шифрование является детерминированным, поэтому шифрование одного и того же блока открытого текста дважды приведет к одному и тому же результату. Из этого следует, что при шифровании каждого отдельного блока в полученном шифротексте могут появиться повторяющиеся паттерны.

Казалось бы, ничего страшного, но наличие таких повторений приводит к множеству проблем. Наиболее очевидная из них — утечка данных, касающихся открытого текста. Самым известным примером этого является пингвин ECB, изображенный на рис. 4.7.

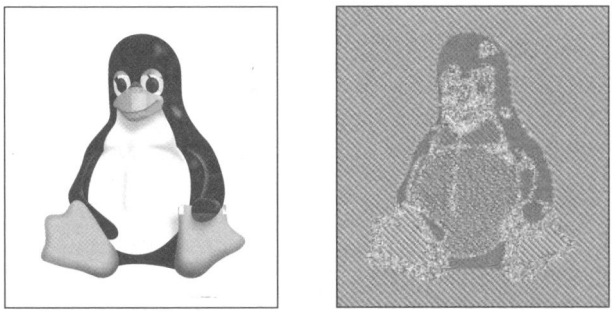


Рис. 4.7. Знаменитый пингвин ECB — это шифрование изображения пингвина в режиме электронной кодовой книги (ECB). Поскольку ECB не скрывает повторяющиеся паттерны, по шифротексту можно догадаться, что было зашифровано (изображение взято из «Википедии»)

Для безопасного шифрования открытого текста размером более 128 бит существуют более совершенные режимы, которые рандомизируют результат. Один из наиболее популярных режимов шифрования AES — это *режим сцепления блоков шифротекста* (Cipher Block Chaining, CBC). Он применим к любому детерминированному блочному шифру, не только к AES, поскольку для рандомизации шифрования берется дополнительное значение — *вектор инициализации* (initialization vector, IV). Он должен иметь длину, равную размеру блока (16 байт в AES), а также случайный и непредсказуемый вид.

Для шифрования в режиме CBC необходимо сгенерировать случайный IV из 16 байт (как это сделать, рассказано в главе 8), выполнить операцию XOR между сгенерированным IV и первыми 16 байтами открытого текста, а затем зашифровать результат. Таким образом происходит эффективная рандомизация шифрования. Действительно, если зашифровать один и тот же открытый текст дважды, но использовать при этом разные IV, то результатом будут два разных шифротекста.

Если нужно зашифровать еще один открытый текст, то сначала выполняется операция XOR между предыдущим шифротекстом, взятым здесь в качестве IV, и шифруемым блоком открытого текста, что рандомизирует следующий блок шифрования. Помните, что при шифровании чего-либо необходимо обеспечить непредсказуемость, равную той, которую дает генератор случайных чисел и которую мы применили для создания реального IV. Шифрование в режиме CBC иллюстрирует рис. 4.8.

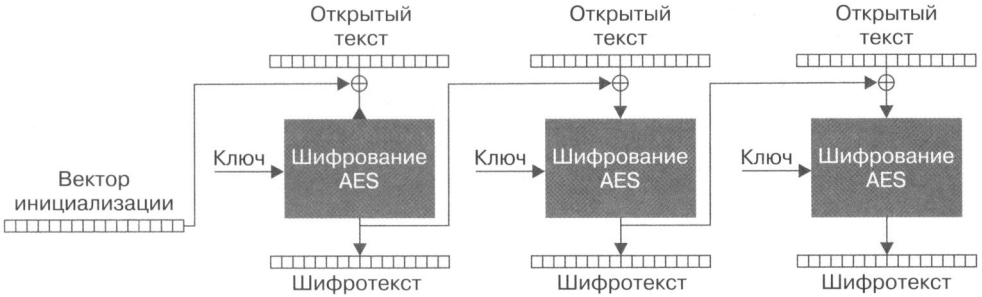


Рис. 4.8. Режим шифрования CBC в AES. При шифровании добавляем случайный вектор инициализации (IV) к заполненному открытому тексту, разбитому на несколько блоков по 16 байт

Для расшифровки в режиме CBC необходимо обратить все шаги шифрования. Поскольку требуется IV, он должен быть передан в открытом виде вместе с шифротекстом. Не стоит опасаться утечки информации зашифрованного текста при обнаружении значения вектора инициализации, так как он имеет случайный вид. Расшифровка в режиме CBC показана на рис. 4.9.

В криптографии широко применяются такие дополнительные параметры, как векторы инициализации. Однако зачастую они представляют собой плохо изученные элементы и, следовательно, крупный источник уязвимостей. В режиме CBC вектор инициализации должен иметь *уникальный* и *непредсказуемый* характер,

то есть неповторимый и действительно случайный вид. Однако эти требования могут не выполняться по ряду причин. Из-за того что разработчики часто испытывают сложности при работе с IV, некоторые криптографические библиотеки убрали возможность вручную задавать его при шифровании в режиме CBC и автоматически генерируют случайным образом.

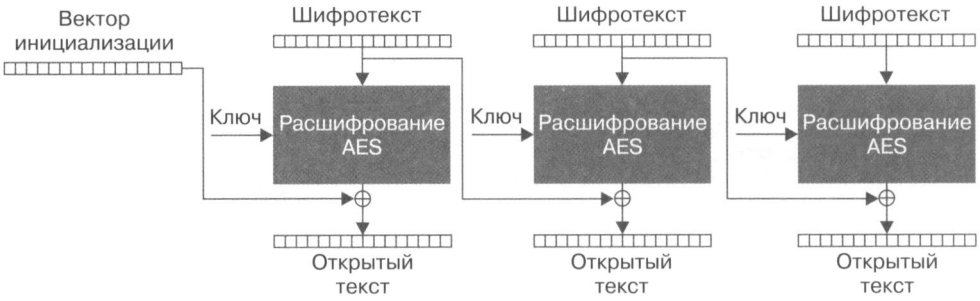


Рис. 4.9. Режим CBC в AES. Для расшифровки требуется соответствующий вектор инициализации

ВНИМАНИЕ

Когда вектор инициализации повторяется или его можно угадать, шифрование снова становится детерминированным, что делает возможным успешное проведение ряда хитроумных атак. Так было в случае знаменитой атаки BEAST (Browser Exploit Against SSL/TLS) на протокол TLS. Отмечу также, что другие алгоритмы могут предъявлять иные требования к IV. Поэтому всегда важно читать инструкцию. Опасность кроется в мелком шрифте.

Замечу, что только режима шифрования и дополнения недостаточно для того, чтобы сделать шифр пригодным для использования. В следующем разделе вы узнаете почему.

4.4. ОТСУТСТВИЕ ПОДЛИННОСТИ И РЕШЕНИЕ ПРОБЛЕМЫ В ВИДЕ AES-CBC-HMAC

До сих пор нам не удалось устранить один фундаментальный недостаток — шифротекст, как и вектор инициализации в случае CBC, может быть изменен злоумышленником. Не существует механизма защиты целостности, который бы это предотвратил! Изменения в шифротексте или IV могут привести к неожиданным изменениям при расшифровке. Например, в AES-CBC (AES с режимом CBC) злоумышленник может повлиять на определенные биты открытого текста, переключив их в соответствующих ему векторе инициализации и шифротексте (рис. 4.10).

Следовательно, шифр или режим шифрования нельзя использовать в чистом виде. В них отсутствует какая-либо защита целостности, которая гарантировала бы, что шифротекст и связанные с ним параметры (в данном случае вектор инициализации) не могут быть изменены без оповещения об этом.

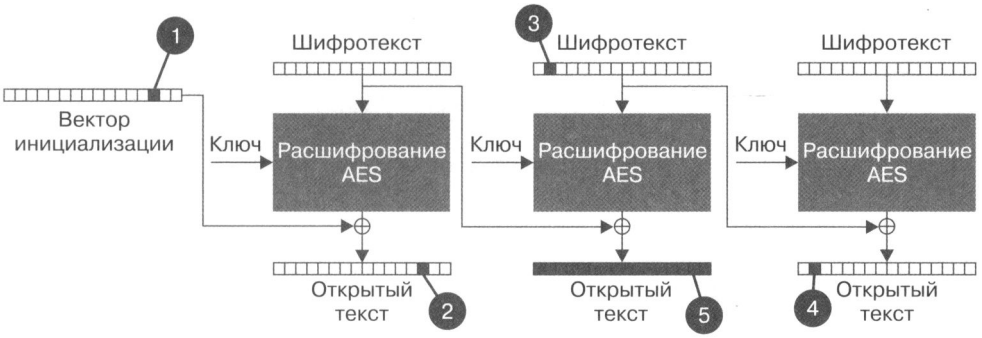


Рис. 4.10. Злоумышленник, перехвативший шифротекст AES-CBC, может сделать следующее. Поскольку вектор инициализации находится в открытом доступе, переключение бита в нем, например, с 1 на 0 *(1)* переключает бит первого блока открытого текста *(2)*. Изменения битов могут происходить также в блоках шифротекста *(3)*. Они влияют на следующий блок расшифрованного открытого текста *(4)*. Важно отметить, что вмешательство непосредственно в блок шифротекста искажает его расшифровку *(5)*

Для предотвращения изменений шифротекста можно задействовать знакомые нам из главы 3 *имитовставки*. При работе с AES-CBC для обеспечения целостности данных обычно используется HMAC в сочетании с хеш-функцией SHA-256. Мы применяем MAC как на шифротексте, так и на векторе инициализации после дополнения и шифрования открытого текста, в противном случае злоумышленник может изменить IV, не будучи пойманным.

ВНИМАНИЕ

Данная конструкция называется Encrypt-then-MAC. Альтернативных вариантов, таких как MAC-then-Encrypt, стараются избегать, поскольку они иногда приводят к попыткам проведения хитроумных атак, например знаменитой атаки Вудене Padding Oracle.

Созданный аутентификационный тег может быть передан вместе с IV и шифротекстом. Как правило, все эти элементы объединены между собой (рис. 4.11). Кроме того, лучше всего использовать разные ключи для AES-CBC и HMAC.

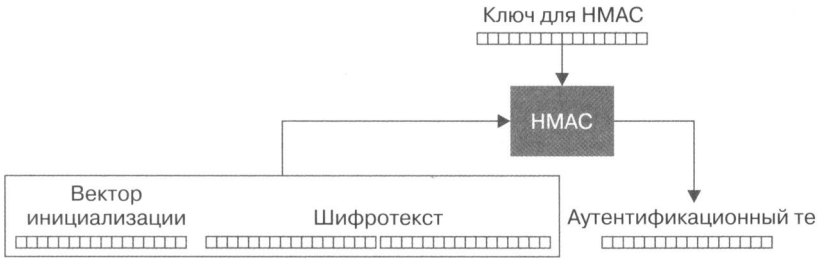


Рис. 4.11. Конструкция AES-CBC-HMAC выдает три аргумента, которые обычно объединяются в следующем порядке: открытый вектор инициализации, шифротекст и аутентификационный тег

Перед расшифровкой тег проверяется (за постоянное время, как вы узнали из главы 3). Комбинация всех этих алгоритмов называется *AES-CBC-HMAC*. Она была одним из наиболее широко используемых режимов аутентифицированного шифрования до появления более современных конструкций типа «все в одном».

ВНИМАНИЕ

AES-CBC-HMAC не самая удобная для разработчиков конструкция. Она зачастую плохо реализована и имеет ряд опасных подводных камней, появляющихся при неправильном использовании (например, IV каждого шифрования должен иметь непредсказуемый вид). Я выделил на эту конструкцию несколько страниц из-за ее широкого применения и работоспособности. Тем не менее стоит применять вместо нее более современные конструкции, представленные далее.

4.5. КОМПЛЕКСНЫЕ КОНСТРУКЦИИ. АУТЕНТИФИЦИРОВАННОЕ ШИФРОВАНИЕ

История шифрования не отличается красотой. Мало того что люди плохо осознавали опасность шифрования без аутентификации, так еще и неправильное применение аутентификации было систематической ошибкой разработчиков. По этой причине появилось множество исследований, направленных на стандартизацию комплексных конструкций, которые упрощали бы процесс шифрования для разработчиков. В оставшейся части этого раздела я познакомлю вас с этой новой концепцией, а также двумя широко распространенными стандартами: AES-GCM и ChaCha20-Poly1305.

4.5.1. Что такое аутентифицированное шифрование с присоединенными данными

Наиболее современным способом шифрования данных является использование комплексной конструкции под названием «аутентифицированное шифрование с присоединенными данными» (authenticated encryption with associated data, AEAD). Эта конструкция очень близка к AES-CBC-HMAC: она так же обеспечивает конфиденциальность открытых текстов и одновременно отслеживает любые возможные изменения в шифротекстах. Более того, в ней предусмотрена аутентификация *присоединенных данных*.

Присоединенные данные — это необязательный аргумент, который может быть пустым или содержать метаданные, имеющие отношение к шифрованию и расшифрованию открытого текста. Присоединенные данные не шифруются и либо подразумеваются, либо передаются вместе с шифротекстом. При этом размер шифротекста увеличивается по сравнению с открытым текстом, поскольку теперь он содержит дополнительный аутентификационный тег, который обычно добавляется в конец шифротекста.

Для расшифровки нам нужны те же подразумеваемые или передаваемые присоединенные данные (ПД).. В результате мы получаем либо ошибку, указывающую на то, что шифротекст был изменен при передаче, либо исходный открытый текст. На рис. 4.12 показан этот примитив.

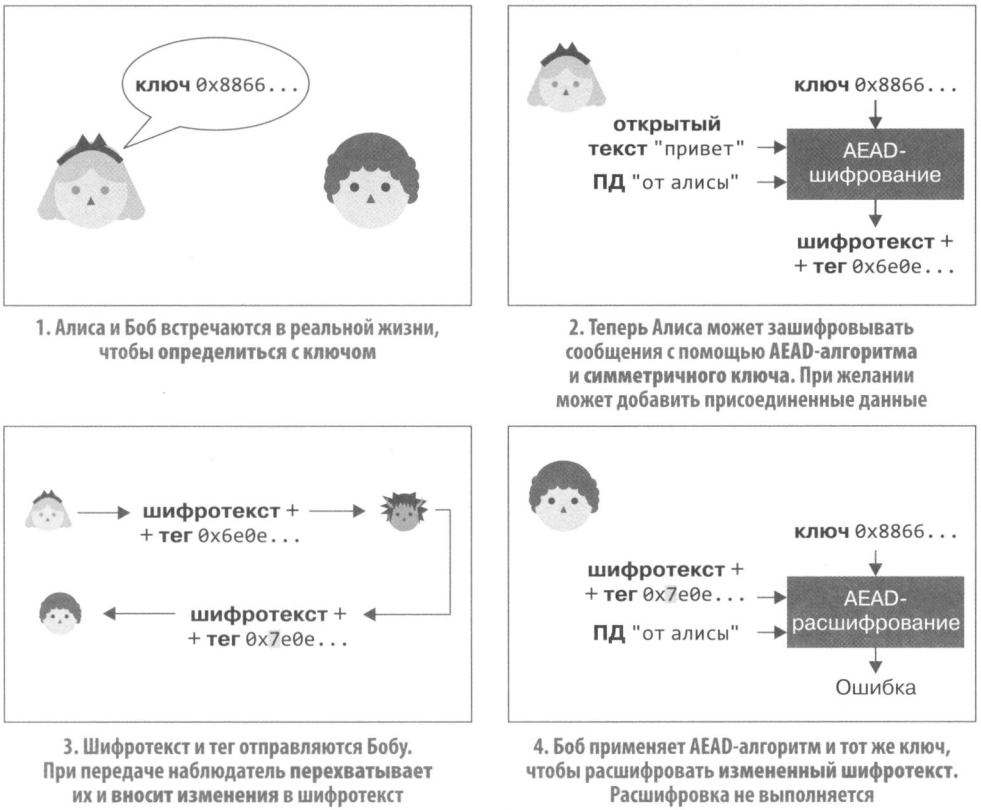


Рис. 4.12. Алиса и Боб встречаются лично, чтобы определиться с общим ключом. Затем Алиса может использовать AEAD-алгоритм и этот ключ для шифрования своих сообщений Бобу. При желании она может аутентифицировать их с помощью присоединенных данных, например указать отправителя сообщения. Получив шифротекст вместе с аутентификационным тегом, Боб может расшифровать его с помощью того же ключа и присоединенных данных. Если эти данные неверны или шифротекст был изменен при передаче, то расшифровка не выполняется

Рассмотрим, как применять *криптографическую библиотеку* для шифрования и расшифрования с помощью примитива аутентифицированного шифрования. Для этого используем язык программирования JavaScript и Web Crypto API (официальный интерфейс, поддерживаемый большинством браузеров и предоставляющий низкоуровневые криптографические функции), как показано в листинге 4.1.

Листинг 4.1. Аутентифицированное шифрование с AES-GCM на JavaScript

```
let config = {
    name: 'AES-GCM',
    length: 128
};
let keyUsages = ['encrypt', 'decrypt'];
let key = await crypto.subtle.generateKey(config, false, keyUsages);

let iv = new Uint8Array(12);
await crypto.getRandomValues(iv);

let te = new TextEncoder();
let ad = te.encode("some associated data");
let plaintext = te.encode("hello world");

let param = {
    name: 'AES-GCM',
    iv: iv,
    additionalData: ad
};
let ciphertext = await crypto.subtle.encrypt(param, key, plaintext);

let result = await window.crypto.subtle.decrypt(
    param, key, ciphertext);
new TextDecoder("utf-8").decode(result);
```

Генерирует 128-битный ключ для обеспечения 128 бит защиты

Генерирует случайный 12-байтовый вектор инициализации

Добавляет присоединенные данные, чтобы зашифровать открытый текст. Для расшифровки нужны те же IV и присоединенные данные

Расшифрование вызывает исключение при подделке IV, шифротекста или присоединенных данных

Отмечу, что Web Crypto API является низкоуровневым API и сам по себе не помогает разработчику избежать ошибок. Например, он позволяет указывать IV, что представляет собой опасный паттерн. В листинге 4.1 был использован наиболее распространенный AEAD-алгоритм – AES-GCM. Рассмотрим его более подробно.

4.5.2. AEAD-алгоритм AES-GCM

Наиболее распространенным AEAD-режимом является AES *с режимом Галуа/счетчика* (Galois/Counter Mode, AES-GCM). Он был разработан для обеспечения высокой производительности за счет преимуществ аппаратной поддержки AES и эффективной реализации MAC (GMAC).

С 2007 года AES-GCM включен в специальную публикацию NIST (SP 800-38D) и является основным шифром, используемым в криптографических протоколах, включая несколько версий TLS-протокола, который применяется для защиты соединений с интернет-сайтами. По сути, можно сказать, что AES-GCM шифрует Интернет.

AES-GCM сочетает в себе режим счетчика (Counter Mode, CTR) и имитовставку GMAC. Сначала посмотрим, как режим CTR и AES работают вместе (рис. 4.13).

AES-CTR задействует AES для шифрования не открытого текста, а nonce, сцепленного с числом начиная с 1. Дополнительный аргумент «nonce как однократно используемое число» имеет ту же цель, что и IV: он позволяет режиму шифрования рандомизировать AES-шифрование. Однако требования к нему несколько отличаются от требований к IV. Для nonce обязательна уникальность, а вот его предсказуемость *неважна*. Результат шифрования этого 16-байтового блока называется *потоком ключей*. После выполнения операции XOR между потоком ключей и открытым текстом получается итоговый шифр.

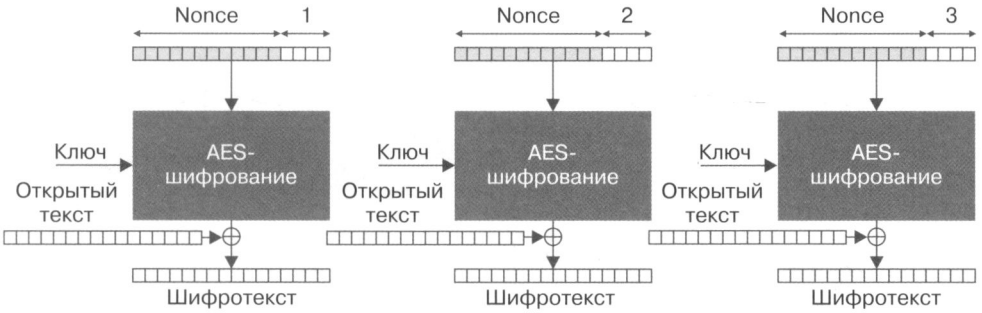


Рис. 4.13. Алгоритм AES-CTR, сочетающий в себе шифр AES с режимом счетчика (режимом CTR). Уникальный nonce объединяется со счетчиком и зашифровывается, в результате чего получается поток ключей. Затем между ним и фактическими байтами открытого текста выполняется операция XOR

ПРИМЕЧАНИЕ

Как и векторы инициализации, nonce является общим термином в криптографии и встречается в различных криптографических примитивах. Несмотря на то что из названия следует, что аргументы nonce не должны повторяться, к ним могут предъявляться различные требования. И как правило, важнее именно то, что написано в руководстве, а не то, что подразумевает название аргумента. Фактически nonce AES-GCM иногда можно назвать IV.

Nonce в AES-CTR имеет размер 96 бит (12 байт) и занимает большую часть 16-байтового шифрующего блока. Оставшиеся 32 бита (4 байта) выполняют функцию счетчика, который начинается с 1 и увеличивается с шифрованием каждого последующего блока, пока не достигнет максимального значения $2^{4\times8} - 1 = 4\ 294\ 967\ 295$. Следовательно, максимум 4 294 967 295 блоков по 128 бит (то есть менее 69 Гбайт) могут быть зашифрованы одним и тем же nonce.

Если один и тот же nonce используется дважды, то создается один и тот же поток ключей. Операция XOR между двумя шифротекстами аннулирует этот поток ключей, что позволит восстановить результат операции XOR, выполненной между двумя соответствующими открытыми текстами. Это может оказаться губительным, особенно если у злоумышленника есть информация о содержимом одного из двух открытых текстов.

На рис. 4.14 показан интересный аспект режима CTR: он не требует дополнения. CTR превращает блочный шифр (AES) в потоковый. Он шифрует открытый текст побайтно.

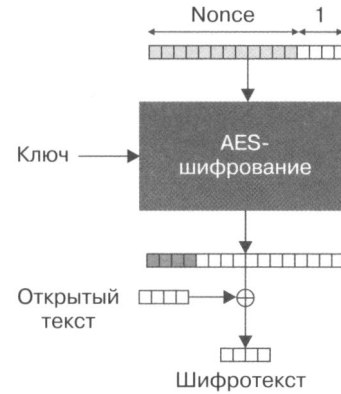


Рис. 4.14. Если поток ключей AES-CTR длиннее, чем открытый текст, он усекается до длины открытого текста и только потом между ним и открытым текстом выполняется операция XOR. Это позволяет AES-CTR работать без использования дополнения

ПОТОКОВЫЕ ШИФРЫ

Потоковые шифры — это другая категория шифров. Они отличаются от блочных тем, что их можно использовать непосредственно для шифрования открытого текста путем применения операции XOR между ними и потоком ключей. При этом не требуется ни дополнения, ни режима шифрования, что позволяет шифротексту иметь ту же длину, что и открытый текст.

На практике разница между этими двумя категориями шифров невелика, поскольку блочные шифры легко преобразуются в потоковые с помощью режима CTR. Однако в теории преимущество отдается блочным шифрам, поскольку они могут быть полезны при построении других категорий примитивов, как мы видели в главе 2, посвященной хеш-функциям.

Здесь также стоит отметить, что по умолчанию шифрование не скрывает или плохо скрывает длину шифруемого элемента. В связи с этим сжатие перед шифрованием может привести к атакам, если злоумышленник сможет повлиять на часть шифруемого элемента.

Второй частью AES-GCM является *GMAC*. Она представляет собой имитовставку, построенную на основе хеша с ключом — так называемого *GHASH*. Говоря техническим языком, GHASH — это почти XOR-универсальная хеш-функция (almost XORed universal hash, AXU), которую называют также *функцией с непредсказуемым отличием* (difference unpredictable function, DUF). Требования к такой функции не такие строгие, как к хешу. Например, AXU не обязательно должна быть устойчива к коллизиям. Благодаря этому GHASH (рис. 4.15) может быть значительно быстрее.

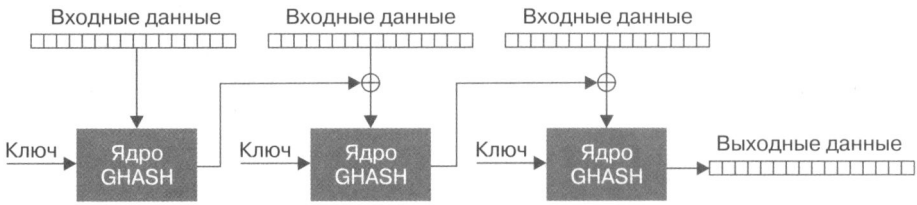


Рис. 4.15. GHASH принимает ключ и блок за блоком поглощает входные данные, чем-то напоминая режим CBC. В результате получается дайджест из 16 байт

Чтобы захешировать что-либо с помощью GHASH, мы разбиваем входные данные на блоки по 16 байт и затем хешируем их, как в режиме CBC. Поскольку этот хеш принимает на вход ключ, теоретически он может быть использован в качестве имитовставки, но только один раз (иначе алгоритм ломается), то есть это *одноразовая имитовставка*. Поскольку для нас это не идеальный вариант, мы применяем технику (заслуга Вегмана и Картера) для преобразования GHASH в *многоразовую имитовставку* (рис. 4.16).

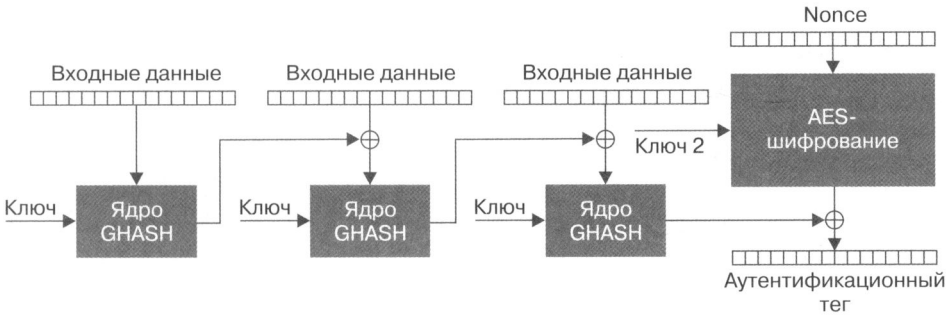


Рис. 4.16. GMAC хеширует входные данные с помощью GHASH и ключа, затем шифрует их с помощью другого ключа и AES-CTR для получения аутентификационного тега

Фактически GMAC — это шифрование выходных данных GHASH с помощью AES-CTR и другого ключа. Опять же уникальность nonce является обязательным требованием, иначе умные злоумышленники смогут восстановить ключ аутентификации GHASH. Это может стать катастрофой, так как позволит без труда подделывать аутентификационные теги.

AES-GCM можно рассматривать как переплетенную комбинацию режима CTR и GMAC, аналогичную рассмотренной ранее конструкции Encrypt-then-MAC. Весь алгоритм проиллюстрирован на рис. 4.17.

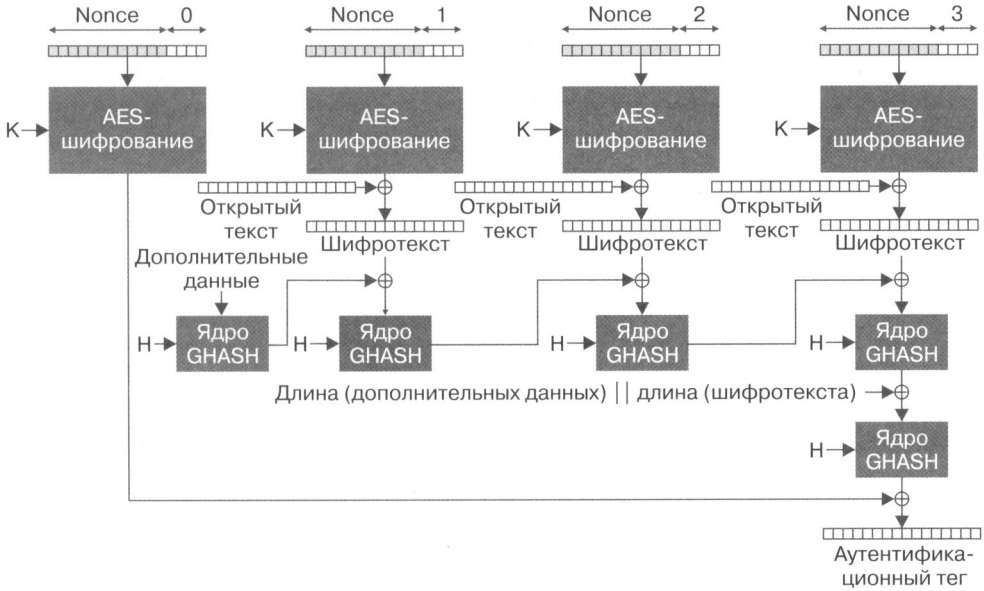


Рис. 4.17. Принцип работы алгоритма AES-GCM заключается в том, что он использует AES-CTR с симметричным ключом K для шифрования открытого текста и GMAC — для аутентификации присоединенных данных и шифротекста с помощью ключа аутентификации H

В начале шифрования счетчик имеет значение 1, так как значение 0 оставляется для шифрования аутентификационного тега, создаваемого GHASH. В свою очередь, GHASH принимает независимый ключ H, который представляет собой результат шифрования блока, состоящего из одних нулей, с ключом K. Таким образом, нет необходимости держать два разных ключа, поскольку ключа K достаточно для получения другого.

Как говорилось ранее, 12-байтовый аргумент nonce в AES-GCM должен быть уникальным и, следовательно, никогда не повторяться. Обратите внимание на то, что ему не обязательно иметь случайный характер. В связи с этим некоторые предпочитают использовать его в качестве *счетчика*, начиная с 1 и увеличивая значение на каждом следующем этапе шифрования. В этом случае необходимо задействовать криптографическую библиотеку, позволяющую пользователю выбирать nonce. Таким образом можно зашифровать $2^{12 \times 8} - 1$ сообщение до достижения максимального значения nonce. Достаточно сказать, что на практике такого количества сообщений достичь невозможно.

В то же время наличие счетчика означает необходимость сохранения *состояния*. Если машина выйдет из строя в неподходящий момент, то возможно повторное использование nonce. По этой причине иногда предпочтительнее иметь *случайный nonce*. Некоторые библиотеки не позволяют разработчикам выбирать значение nonce и генерируют его случайным образом. Это позволяет избежать повторения с такой высокой вероятностью, что практически исключается вероятность возникновения данной проблемы. Тем не менее чем больше сообщений шифруется, тем больше аргументов nonce используется и тем выше вероятность возникновения коллизии. В связи с ограниченностью дней рождения, о которой мы говорили в главе 2, при случайной генерации nonce рекомендуется не шифровать одним и тем же ключом более $2^{96/3} \approx 2^{32}$ сообщений.

КРИПТОСТОЙКОСТЬ ЗА ПРЕДЕЛАМИ ОГРАНИЧЕННОСТИ ДНЕЙ РОЖДЕНИЯ

Согласитесь, 2^{30} сообщений — это довольно большое количество. В теории его невозможно достичь в большинстве допустимых сценариев. Однако реальная криптография часто расширяет границы того, что считается разумным. Некоторые долгоживущие системы должны шифровать огромное количество сообщений в секунду и в итоге достигают этих пределов. Например, Visa обрабатывает 150 млн транзакций в день. Если ей потребуется шифровать эти транзакции уникальным ключом, то всего за неделю она достигнет предела в 2^{30} сообщений. В таких крайних случаях решением проблемы может стать *смена ключей* (замена ключа, используемого для шифрования). Существует также область исследований под названием «криптостойкость за пределами ограниченности дней рождения» (beyond birthday-bound security), целью которой является увеличение максимального числа сообщений, которые могут быть зашифрованы одним и тем же ключом.

4.5.3. ChaCha20Poly1305

Второй AEAD-режим, о котором я расскажу, — это *ChaCha20Poly1305*. Он представляет собой комбинацию двух алгоритмов: потокового шифра ChaCha20 и MAC Poly1305. Оба были разработаны по отдельности Дэниелом Дж. Бернштейном и представляли собой алгоритмы с высокой скоростью работы в программном обеспечении в противовес медленному при отсутствии аппаратной поддержки AES. В 2013 году компания Google стандартизировала ChaCha20Poly1305 и начала применять его в мобильных телефонах на базе Android с процессорами низкого класса. В настоящее время он широко используется в таких интернет-протоколах, как OpenSSH, TLS и Noise.

ChaCha20 — это модификация потокового шифра Salsa20, разработанного Дэниелом Дж. Бернштейном примерно в 2005 году. Он был одним из алгоритмов — номинантов конкурса ESTREAM (https://www.ecrypt.eu.org/stream/). Как и во всех потоковых шифрах, в Salsa20 создается *поток ключей* — серия случайных байтов, длина которой равна длине открытого текста. Затем для создания шифротекста он объединяется с открытым текстом с помощью операции XOR. Для расшифрования тот же алгоритм производит тот же поток ключей, который затем объединяется с шифротекстом посредством операции XOR, в результате чего получается открытый текст. Оба процесса показаны на рис. 4.18.

Для получения множества 64-байтовых блоков потока ключей ChaCha20 генерирует поток ключей, многократно вызывая *блочную функцию*. Она принимает:

- 256-битный (32 байта) ключ, как AES-256;
- 96-битный (12 байт) nonce, как AES-GCM;
- 32-битный (4 байта) счетчик, как AES-GCM.

Процесс шифрования такой же, как в AES-CTR (рис. 4.19).

1. Запускается блочная функция, при каждом запуске значение счетчика увеличивается до тех пор, пока не будет получен достаточный поток ключей.
2. Поток ключей усекается до длины открытого текста.
3. Между потоком ключей и открытым текстом выполняется операция XOR.

Верхняя граница счетчика позволяет с помощью ChaCha20 зашифровать столько же сообщений, сколько и с помощью AES-GCM (он параметризуется аналогичным значением nonce). Поскольку размер выхода этой блочной функции значительно больше, то и размер сообщения, которое можно зашифровать, тоже больше. Таким образом, можно зашифровать сообщение размером $2^{32} \times 64$ байта ≈ 274 Гбайт. Повторное использование nonce при шифровании открытого текста создает такие же проблемы, как и в случае с AES-GCM. Наблюдатель может получить результат операции XOR между двумя открытыми текстами, проведя ее между двумя шифротекстами, а также восстановить ключ аутентификации для nonce. Это серьезные уязвимости, так как они дают злоумышленнику возможность подделывать сообщения!

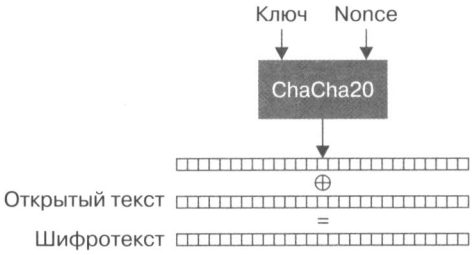


Рис. 4.18. ChaCha20 использует в работе симметричный ключ и уникальный nonce. С их помощью генерируется поток ключей, который объединяется с открытым текстом (или шифротекстом) операцией XOR, в результате чего получается шифротекст (или открытый текст). При таком шифровании сохраняется длина, поскольку длина шифротекста и открытого текста одинаковая

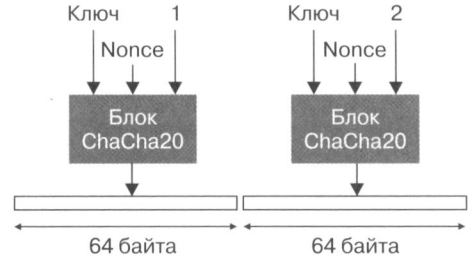


Рис. 4.19. Поток ключей ChaCha20 создается посредством вызова внутренней блочной функции до тех пор, пока не будет получено достаточное количество байтов. Одна блочная функция создает случайный поток ключей размером 64 байта

РАЗМЕР NONCE И СЧЕТЧИКОВ

Размеры nonce и счетчиков (на самом деле) не всегда и не везде одинаковы как для AES-GCM, так и для ChaCha20Poly1305, но это рекомендуемые значения в принятых стандартах. Тем не менее некоторые криптографические библиотеки допускают различные размеры nonce, а часть приложений увеличивают размер счетчика или nonce для шифрования больших сообщений или их большого количества. При этом увеличение размера одного аргумента алгоритма обязательно происходит за счет уменьшения размера другого.

Для предотвращения этого и обеспечения возможности шифрования большого количества сообщений одним ключом существуют другие стандарты, например XChaCha20Poly1305. В них увеличение размера nonce не затрагивает остальные аргументы алгоритма, что важно в тех случаях, когда nonce генерируется случайным образом, а не служит отслеживаемым в системе счетчиком.

Внутри блочной функции ChaCha20 формируется состояние (рис. 4.20).

Затем это состояние преобразуется в 64 байта потока ключей путем 20-кратной итерации раундовой функции (отсюда и число 20 в названии алгоритма). Здесь все аналогично раундовой функции в AES: она вызывает *функцию Quarter Round* (QR) четыре раза за раунд, каждый раз воздействуя на разные слова внутреннего состояния в зависимости от четности и нечетности номера раунда (рис. 4.21).

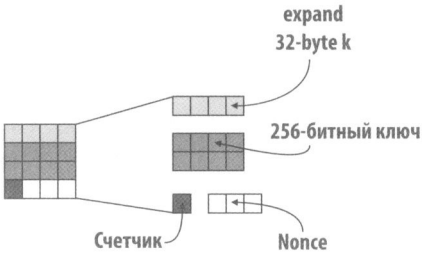


Рис. 4.20. Состояние блочной функции ChaCha20 формируется из 16 слов (обозначены квадратами) по 32 бита каждое. В первой строке хранится константа, во второй и третьей — 32-байтовый симметричный ключ, в следующем слове — 4-байтовый счетчик, а в последних трех — 12-байтовое значение nonce

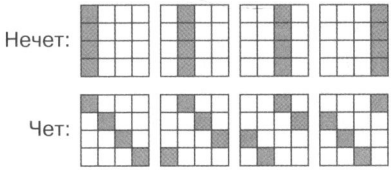


Рис. 4.21. Раунд в ChaCha20 затрагивает все слова, содержащиеся в состоянии. Поскольку функция Quarter Round принимает только четыре аргумента, то для изменения всех 16 слов состояния ее необходимо вызвать не менее четырех раз на разных словах (выделены серым цветом)

Функция QR принимает четыре различных аргумента и обновляет их, используя только операции сложения, сдвига и XOR, отсюда и другое ее название — потоковый шифр *ARX* (от Add, Rotate и XOR). Таким образом, ChaCha20 считается чрезвычайно простым и быстрым шифром в программной реализации.

Poly1305 — это имитовставка, созданная по методике Вегмана — Картера. Она очень похожа на GMAC, о котором мы говорили ранее. Принцип ее работы изображен на рис. 4.22.

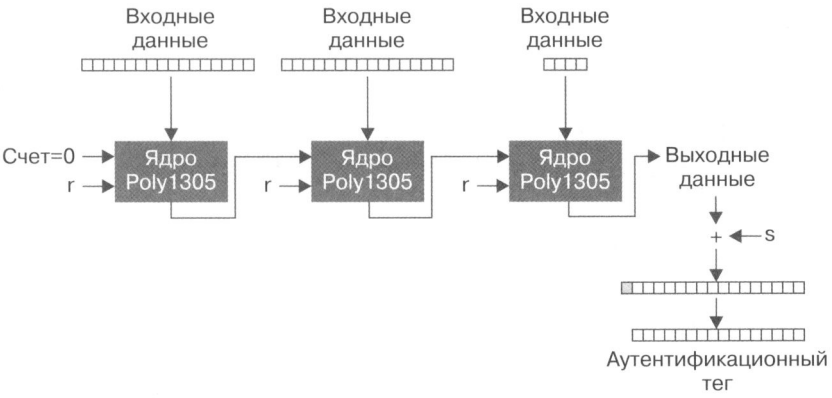


Рис. 4.22. Ядро функции Poly1305 поглощает входные данные по одному блоку, принимая дополнительный счетчик с начальным значением 0 и ключ аутентификации r. Выходные данные служат счетчиком для следующего вызова ядра функции. В конце шифрования выходные данные складываются со случайной величиной s, и таким образом получается аутентификационный тег

На рисунке *r* можно рассматривать как ключ аутентификации схемы, то есть как ключ аутентификации *H* в GMAC. Аргумент *s* шифрует результат и тем самым делает имитовставку безопасной для многократного использования, поэтому он должен иметь уникальное значение при каждом своем применении.

Ядро функции Poly1305 выполняет аутентификацию сообщения путем смешивания его с ключом, содержащим счетчик, начальное значение которого равно 0. Операции представляют собой простое умножение по модулю константы *P*.

ПРИМЕЧАНИЕ

Очевидно, что в нашем тексте не хватает многих деталей. Я редко говорю о том, как кодировать данные или как заполнять некоторые аргументы перед выполнением действия. Эти особенности реализации для нас неважны, поскольку мы пытаемся получить интуитивное представление о том, как все это работает.

В итоге мы можем использовать ChaCha20 и счетчик со значением 0, чтобы сгенерировать поток ключей и получить 16-байтовые значения *r* и 16-байтовые значения *s*, необходимые для Poly1305. Полный AEAD-шифр показан на рис. 4.23.

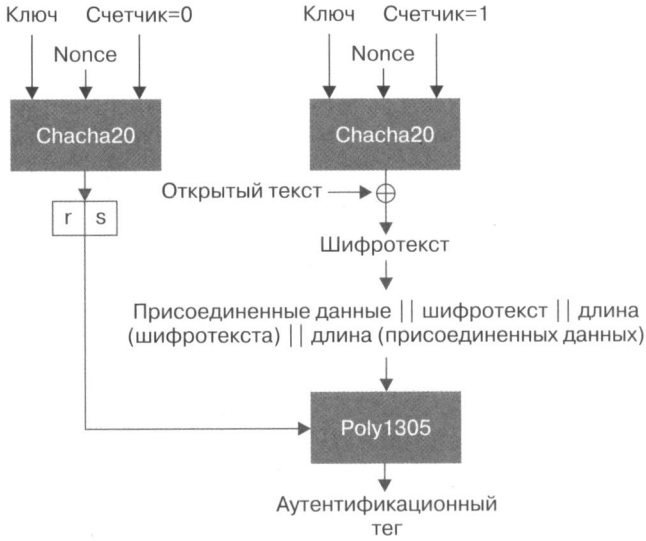


Рис. 4.23. ChaCha20 Poly1305 применяет шифр ChaCha20 для шифрования открытого текста и получения ключей, необходимых для работы имитовставки Poly1305. Затем с помощью Poly1305 выполняется аутентификация шифротекста и присоединенных к нему данных

В самом начале посредством ChaCha20 мы получаем аутентификационные секреты *r* и *s*, необходимые для работы Poly1305. После этого значение счетчика увеличивается на 1 и ChaCha20 шифрует открытый текст. Затем присоединенные данные и шифротекст (и их длины) передаются в Poly1305 для создания аутентификационного тега.

Процесс расшифровки аналогичен: сначала ChaCha20 по полученному тегу проверяет подлинность шифротекста и присоединенных к нему данных. Затем происходит расшифровка шифротекста.

4.6. ДРУГИЕ ВИДЫ СИММЕТРИЧНОГО ШИФРОВАНИЯ

Давайте сделаем небольшую паузу и подведем итоги. К этому моменту мы рассмотрели следующие алгоритмы симметричного шифрования:

- *неаутентифицированное шифрование* — AES с режимом шифрования, но без имитовставки. На практике использование этого алгоритма небезопасно, так как шифротексты могут быть подделаны;
- *аутентифицированное шифрование* — AES-GCM и ChaCha20Poly1305 — два наиболее распространенных шифра.

На этом главу можно было бы закончить. Однако в реальном мире криптография не всегда сводится к согласованным стандартам — она связана также с *ограничениями* по размеру, скорости, формату и т. д. Исходя из этого, позвольте мне дать краткий обзор других типов симметричного шифрования, которые могут пригодиться в тех случаях, когда не подходят AES-GCM и ChaCha20Poly1305.

4.6.1. Обертка ключа

Одна из проблем AEAD-алгоритмов, основанных на nonce, заключается в том, что все они требуют наличия nonce, занимающего дополнительное место. Замечу, что при шифровании ключа рандомизация может и не потребоваться, поскольку зашифрованное уже носит случайный характер и, вероятнее всего, не будет иметь дублей (а если и будет, то это не страшно). Одним из известных стандартов обертки ключей (key wrapping) является специальная публикация NIST 80038F. «Рекомендация по режимам шифрования блочных шифров: метод обертки ключа». Эти алгоритмы симметричного шифрования ключей не используют дополнительный nonce или IV, а рандомизируют шифрование в зависимости от того, что именно они шифруют. Благодаря этому им не нужно хранить дополнительный nonce или IV рядом с шифротекстами.

4.6.2. Аутентифицированное шифрование, устойчивое к некорректному применению nonce

В 2006 году Филипп Рогауэй опубликовал новый алгоритм обертки ключа под названием *«синтетический вектор инициализации»* (synthetic initialization vector, SIV). Он отметил, что SIV полезен не только для шифрования ключей, но и в качестве общей AEAD-схемы, которая более терпима к повторному применению nonce. Как вы узнали из этой главы, повторное использование nonce в AES-GCM или ChaCha20Poly1305 может привести к катастрофическим последствиям. В таком случае не только раскрывается результат операции XOR между двумя открытыми

текстами, но и появляется возможность восстановить ключ аутентификации и подделать шифрование сообщений.

Смысл алгоритма, устойчивого к некорректному использованию nonce, заключается в том, что результат шифрования двух открытых текстов с помощью одного и того же nonce позволяет узнать лишь одно: равны эти два открытых текста или нет. Такое решение нельзя назвать первоклассным, но оно явно лучше утечки ключа аутентификации. Данная схема вызвала большой интерес у криптографов и впоследствии была стандартизирована в RFC 8452. «AES-GCM-SIV: Аутентифицированное шифрование, устойчивое к некорректному использованию nonce». Суть схемы SIV заключается в том, что nonce в AEAD генерируется на основе самого открытого текста, что делает крайне маловероятным шифрование двух разных открытых текстов под одним и тем же nonce.

4.6.3. Шифрование диска

Шифрование памяти ноутбука или мобильного телефона имеет ряд серьезных ограничений: оно должно быть быстрым, чтобы этого не замечал пользователь, и может выполняться только на месте (для многих устройств важна экономия места). Поскольку шифрование не может расширяться, AEAD-режимы, которым требуются nonce и аутентификационный тег, здесь не подходят. Вместо них применяется неаутентифицированное шифрование.

Для защиты от *атак переключением битов* (bitflip attacks) большие блоки данных (например, размером в тысячи байт) шифруются таким образом, что однократное переключение битов приводит к срыву расшифровки всего блока. Атака в таком случае с большей вероятностью выведет из строя само устройство, чем достигнет своей цели. Такие конструкции называются *широкоблочными шифрами*, однако данный подход получил также название *«аутентификация бедняка»*.

В Linux-системах и некоторых устройствах на базе Android этот подход реализован с помощью Adiantum — стандартизированной Google в 2019 году широкоблочной конструкции, обернутой в шифр ChaCha. Тем не менее в большинстве устройств используются неидеальные решения. Например, и Microsoft, и Apple применяют не аутентифицированный и не широкоблочный шифр AES-XTS.

4.6.4. Шифрование базы данных

Шифрование данных, находящихся в базе данных, — сложная задача. Поскольку суть этой процедуры заключается в предотвращении утечки данных из базы, ключ, используемый для шифрования и расшифровки, должен храниться вдали от сервера базы данных. Так как клиенты не имеют самих данных, они сильно ограничены в возможности делать запросы к ним.

Простейшее решение этой задачи — *прозрачное шифрование данных* (transparent data encryption, TDE), которое предполагает примитивное шифрование выбранных столбцов. Данная технология хорошо работает в некоторых сценариях, хотя и требует тщательной проверки подлинности присоединенных данных, идентифицирующих шифруемые строку и столбец. В противном случае существует риск подмены зашифрованного содержимого. Тем не менее поиск в зашифрованных

данных невозможен, поэтому в запросах приходится использовать незашифрованные столбцы.

Шифрование с функцией поиска (searchable encryption) — область исследований, направленная на решение этой проблемы. Однако, несмотря на множество предложенных схем, похоже, панацеи не существует: эти схемы подразумевают разные уровни «поисковости», а также разное снижение уровня криптостойкости. Например, слепое индексирование просто позволяет искать точные совпадения, а шифрование с сохранением и раскрытием порядка — упорядочить результаты. В итоге к безопасности этих решений нужно подходить очень внимательно, поскольку их применение — это действительно компромисс.

РЕЗЮМЕ

- Шифрование (или симметричное шифрование) — это криптографический примитив, который может быть использован для защиты конфиденциальности данных. Защита основывается на симметричном ключе, который должен оставаться в тайне.
- Криптостойким может считаться только симметричное шифрование с аутентификацией (именно поэтому мы называем его аутентифицированным шифрованием), поскольку в противном случае существует риск подделки шифротекстов.
- Аутентифицированное шифрование может быть построено на основе алгоритма симметричного шифрования с помощью имитовставки. Однако лучше всего использовать алгоритм аутентифицированного шифрования с присоединенными данными (AEAD), так как он представляет собой комплексную конструкцию, которую сложнее применить неправильно.
- Две стороны могут задействовать аутентифицированное шифрование для сокрытия своих сообщений при условии, что им известен один и тот же симметричный ключ.
- AES-GCM и ChaCha20Poly1305 — два наиболее распространенных AEAD-алгоритма. В наше время они используются в большинстве приложений.
- Повторное применение nonce нарушает аутентификацию AES-GCM и ChaCha20Poly1305. Такие схемы, как AES-GCM-SIV, устойчивы к некорректному использованию nonce, а шифрование ключей позволяет избежать этой проблемы, поскольку в таком случае в nonce нет необходимости.
- Криптография в реальном мире связана с различными ограничениями, и не во всех сценариях AEAD-алгоритмы будут оптимальным решением. Это касается, например, шифрования баз данных или дисков, которые требуют разработки новых конструкций.

5 Обмен ключами

В этой главе

- ✓ Суть и польза обмена ключами.
- ✓ Алгоритм обмена ключами Диффи — Хеллмана и его вариация на эллиптических кривых.
- ✓ Обеспечение безопасности при обмене ключами.

Мы входим в область *асимметричной криптографии*, также называемой *криптографией с открытым ключом*. Первым асимметричным криптографическим примитивом, который мы разберем, будет *обмен ключами*, суть которого, как следует из названия, заключается в обмене ключами. Например, Алиса посылает ключ Бобу, а Боб посылает ключ Алисе. Таким образом два собеседника согласовывают общий секрет, который будет использоваться для шифрования сообщений при помощи алгоритма аутентифицированного шифрования.

ВНИМАНИЕ

Как говорилось во введении, асимметричная криптография гораздо теснее симметричной связана с математикой, поэтому у некоторых читателей следующие главы могут вызвать небольшие затруднения. Не падайте духом! То, что вы узнаете из этой главы, пригодится вам для понимания работы многих других похожих примитивов.

ПРИМЕЧАНИЕ

Чтобы лучше разобраться в материале этой главы, рекомендую прочитать главу 3, посвященную имитовставкам, и главу 4 об аутентифицированном шифровании.

5.1. ЧТО ТАКОЕ ОБМЕН КЛЮЧАМИ

Начнем со сценария, который позволит понять, как обмен ключами может помочь в самых простых ситуациях. Допустим, Алисе и Бобу нужно пообщаться наедине, но они никогда прежде не связывались друг с другом.

Алиса может использовать примитив аутентифицированного шифрования из главы 4. После того как она сгенерирует симметричный ключ и отправит его Бобу, они оба с его помощью смогут зашифровывать свои сообщения. Но что, если за их беседой пассивно наблюдает злоумышленник? Получается, он имеет доступ к симметричному ключу и может расшифровать всю информацию, которой обмениваются Алиса и Боб! Именно в таком случае обмен ключами может заинтересовать наших героев (и нас в будущем). Посредством обмена ключами можно получить такой симметричный ключ, который пассивный наблюдатель не сможет воспроизвести.

Обмен ключами начинается с того, что и Алиса, и Боб генерируют свою *пару ключей* с помощью алгоритма генерации ключей. В результате каждый из них получает закрытый (или секретный) и открытый ключ. Затем они передают друг другу свои *открытые* ключи, особенность которых заключается в том, что их не нужно скрывать от посторонних глаз. Таким образом, оба собеседника могут использовать открытые ключи друг друга в сочетании со своими закрытыми ключами и тем самым вычислить общий секрет (рис. 5.1).

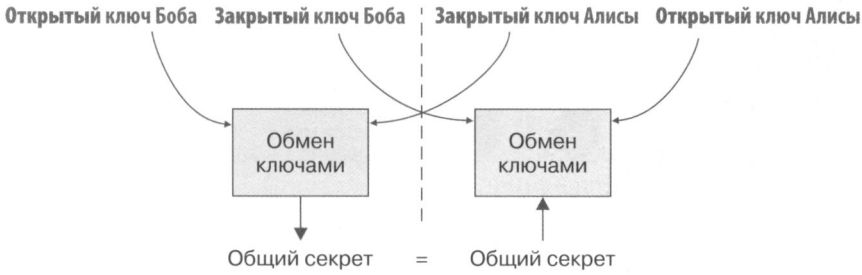


Рис. 5.1. Обмен ключами подразумевает следующее: для создания общего секрета используются открытый ключ вашего собеседника и ваш закрытый ключ. Ваш собеседник может получить тот же общий секрет с помощью вашего открытого ключа и своего закрытого ключа

Получив общее представление о том, как работает обмен ключами, можем вернуться к нашему сценарию. Начав общение с обмена ключами, Алиса и Боб формируют общий секрет, который в дальнейшем будет использоваться как ключ к примитиву аутентифицированного шифрования. Поскольку наблюдающий за этим обменом посредник (man-in-the-middle) не может воспроизвести общий секрет, ему не удастся расшифровать сообщения (рис. 5.2).

Здесь стоит отметить такое условие, как пассивность злоумышленника: *активный* посредник без проблем может перехватить обмен ключами и выдать себя за обе стороны соединения. При такой атаке (рис. 5.3) Алиса и Боб фактически

обмениваются ключами с посредником, причем оба будут думать, что согласовали секретный ключ друг с другом. Причина, по которой это возможно, заключается в том, что ни один из наших героев не имеет возможности проверить, кому действительно принадлежит полученный им открытый ключ. Такой обмен ключами *не подразумевает аутентификации*!



Рис. 5.2. Обмен ключами позволяет двум участникам соединения согласовать секретный ключ. При этом пассивно наблюдающий за обменом посредник не может получить тот же секретный ключ

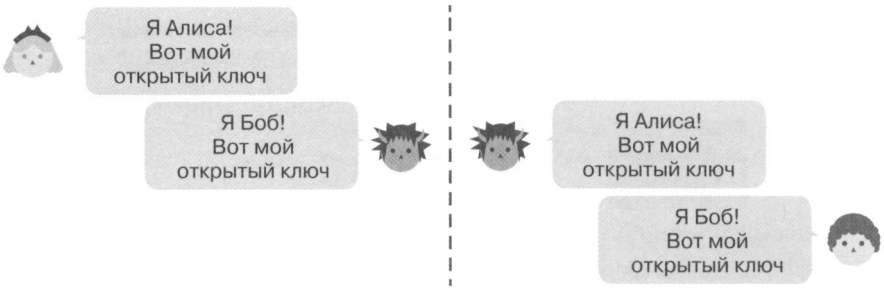


Рис. 5.3. Неаутентифицированный обмен ключами уязвим перед активным злоумышленником, который может выдать себя за обе стороны соединения и выполнить два отдельных обмена ключами

Рассмотрим другой сценарий, в котором обусловлен *аутентифицированный обмен ключами*. Представьте, что вы запускаете сервис, предоставляющий информацию о времени суток. При этом вы не хотите, чтобы какой-нибудь злоумышленник смог изменить ее. Лучшим решением здесь будет проверка подлинности ответов сервиса с помощью имитовставок, о которых мы говорили в главе 3. Требующийся для имитовставки ключ можно просто сгенерировать и вручную передать всем пользователям. Однако тогда получается, что все пользователи имеют один и тот же MAC-ключ и любой из них может когда-нибудь с его помощью провести MITM-атаку против другого пользователя. Конечно, каждому можно дать отдельный ключ, но это тоже неидеальный вариант. Каждый раз, когда к вашему сервису

будет подключаться новый пользователь, вам придется вручную предоставлять новый MAC-ключ и ему, и сервису. Было бы гораздо лучше, если бы не нужно было ничего делать на стороне сервера, не так ли?

Здесь и выручает обмен ключами! Сервис может сгенерировать пару ключей для обмена и предоставить новому пользователю свой открытый ключ. Таким образом, под *аутентифицированным обменом ключами* подразумевается, что пользователи знают открытый ключ сервера, поэтому активному посреднику не удастся выдать себя за эту сторону обмена ключами. Тем не менее злоумышленник может выполнить собственный обмен ключами, поскольку сторона клиента при соединении остается неаутентифицированной. Кстати, когда обе стороны обмена аутентифицированы, мы называем это *взаимно аутентифицированным обменом ключами*.

Данный сценарий встречается очень часто, и обмен ключами позволяет масштабировать защиту при увеличении числа пользователей. Однако этот примитив не сильно поможет при увеличении числа сервисов! Хорошим примером является Интернет. Множество браузеров пытаются обеспечить безопасное соединение с большим количеством сайтов. Представьте, что будет, если в браузере придется жестко кодировать открытый ключ всех сайтов, которые пользователь может однажды посетить, и что произойдет, когда сайтов в Интернете станет еще больше?

Несмотря на полезность обмена ключами, во многих случаях он малоэффективен без своего собрата — *цифровой подписи*. Впрочем, это только анонс. В главе 7 мы подробно поговорим об этом криптографическом примитиве и о том, как он помогает увеличить доверие к системе. На практике обмен ключами редко используется самостоятельно. Как правило, он является составной частью более сложного протокола. Тем не менее в некоторых ситуациях он может быть полезен и сам по себе (например, как вы увидели ранее, против пассивных злоумышленников).

Рассмотрим, как *можно* реализовать обмен ключами на практике. libsodium — это одна из наиболее известных и широко используемых библиотек на языке C/C++. В листинге 5.1 показано, как ее применяют для обмена ключами.

Листинг 5.1. Обмен ключами на языке C

```
unsigned char client_pk[crypto_kx_PUBLICKEYBYTES];
unsigned char client_sk[crypto_kx_SECRETKEYBYTES];
crypto_kx_keypair(client_pk, client_sk);

unsigned char server_pk[crypto_kx_PUBLICKEYBYTES];
obtain(server_pk);

unsigned char decrypt_key[crypto_kx_SESSIONKEYBYTES];
unsigned char encrypt_key[crypto_kx_SESSIONKEYBYTES];

if (crypto_kx_client_session_keys(decrypt_key, encrypt_key,
    client_pk, client_sk, server_pk) != 0) {
    abort_session();
}
```

Генерирует клиентскую пару ключей

Предполагается, что у нас есть способ получить открытый ключ сервера

В соответствии с передовой практикой libsodium получает не один, а два симметричных ключа. Каждый ключ шифрует одно направление

Если открытый ключ некорректный, функция возвращает ошибку

С помощью секретного ключа клиента и открытого ключа сервера выполняется обмен ключами

libsodium скрывает от разработчика многие детали, но при этом предоставляет безопасные для использования интерфейсы. В данном случае применяется

алгоритм обмена ключами X25519, о котором вы узнаете далее. Остальная часть этой главы посвящена различным стандартам для обмена ключами, а также их внутренней работе.

5.2. АЛГОРИТМ ОБМЕНА КЛЮЧАМИ ДИФФИ — ХЕЛЛМАНА

В 1976 году Уитфилд Диффи и Мартин Э. Хеллман написали фундаментальную статью об алгоритме обмена ключами Диффи — Хеллмана (Diffie — Hellman) под названием «Новое направление в криптографии!». Вот это название! Алгоритм Диффи — Хеллмана был первым изобретенным алгоритмом обмена ключами и одним из первых формально описанных криптографических алгоритмов с открытым ключом в целом. В этом разделе я объясню, какие математические концепции лежат в основе данного алгоритма, изложу принцип его работы, а также расскажу о стандартах, описывающих его применение в криптографических приложениях.

5.2.1. Теория групп

Основой для разработки алгоритма обмена ключами Диффи — Хеллмана, а также большей части современной криптографии с открытым ключом является область математики, называемая *теорией групп*. Я постараюсь сделать все возможное, чтобы как можно проще объяснить, как работает этот алгоритм, однако без математики здесь не обойтись. По этой причине сначала рассмотрим, что представляет собой данная теория.

Начнем с очевидного вопроса: что такое *группа*? Это:

- множество элементов;
- специальная бинарная операция, например сложение (+) или умножение (×), определенная на этих элементах.

Если множество и операция удовлетворяют некоторым свойствам, значит, мы имеем группу. А если у нас есть группа, нам доступна магия... (подробнее об этом — позже). Замечу, что алгоритм Диффи — Хеллмана работает в *мультипликативной группе* — такой, где умножение применяется в качестве определенной бинарной операции. В связи с этим в дальнейших пояснениях в качестве примеров будут использоваться именно мультипликативные группы.

Здесь необходимо уточнить. Для того чтобы множество и его операция считались группой, у них должны иметься следующие свойства (как обычно, они показаны на рис. 5.4 для большей наглядности и для того, чтобы вы лучше усвоили теорию).

- *Замкнутость* — результатом операции над двумя элементами является третий элемент того же множества. Например, для двух элементов a и b группы результатом операции ab будет третий элемент этой группы.
- *Ассоциативность* — операции над несколькими элементами подряд могут выполняться в любом порядке. Например, для трех элементов a, b и c группы результатом операций $a(bc)$ и $(ab)c$ будет один и тот же элемент группы.

- *Наличие нейтрального элемента* — в результате операции с таким элементом другой операнд не меняется. Например, в своей мультипликативной группе мы можем определить в качестве нейтрального элемента единицу. Для любого элемента a группы $a \times 1 = a$.
- *Наличие обратного элемента* — существование величины, обратной ко всем элементам группы. Например, для любого элемента a группы существует такой обратный элемент a^{-1} (или $1 / a$), что $aa^{-1} = 1$ (или $a \times 1 / a = 1$).

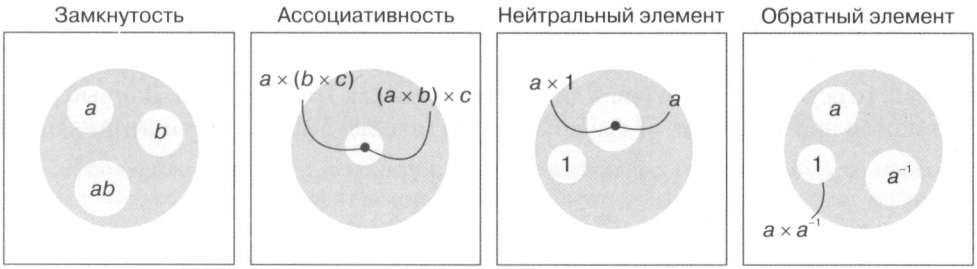


Рис. 5.4. Четыре свойства группы: замкнутость, ассоциативность, наличие нейтрального и обратного элементов

Я понимаю, что мое объяснение группы кажется несколько абстрактным, поэтому давайте посмотрим, что именно является группой в алгоритме Диффи – Хеллмана. Во-первых, алгоритм использует группу, состоящую из множества строго положительных целых чисел: 1, 2, 3, 4... $p - 1$, где p — простое число, а 1 — нейтральный элемент. В разных стандартах указываются различные значения p, но интуитивно понятно, что для обеспечения безопасности группы это должно быть большое простое число.

ПРОСТЫЕ ЧИСЛА

Простое число — это число, которое может делиться только на 1 или на само себя. Первыми простыми числами являются 2, 3, 5, 7, 11 и т. д. В асимметричной криптографии простые числа встречаются повсеместно! И к счастью, у нас есть эффективные алгоритмы для поиска самых больших из них. Для ускорения работы большинство криптографических библиотек ищут *псевдопростые числа* (те, которые с большой вероятностью являются простыми). Интересно, что подобные оптимизации неоднократно подвергались взлому. Наиболее известный случай произошел в 2017 году, когда уязвимость ROCA выявила более миллиона устройств, генерирующих некорректные простые числа для своих криптографических приложений.

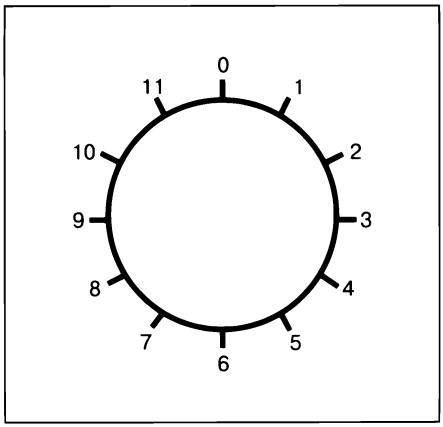
Во-вторых, в алгоритме в качестве специальной операции выступает *модульное умножение*. Прежде чем перейти к модульному умножению, необходимо остановиться на том, что такое *модульная, или модулярная, арифметика*. Речь идет о циклическом повторении отсчета после достижения заданного числа, называемого *модулем*. Например, если мы задаем модуль равным 5, то тем самым говорим, что после 5 отсчет снова начинается с 1, например, 6 становится 1, 7 — 2 и т. д. (Число 5 здесь также принимает значение 0, но, поскольку оно не входит в нашу мультипликативную группу, это не имеет значения.)

Математический способ выражения модульной арифметики заключается в представлении числа как *остатка, взятого от деления по модулю*. К примеру, запишем число 7 в виде остатка от его деления на 5: $7 = 5 \times 1 + 2$. Здесь остаток равен 2, то есть $7 = 2 \bmod 5$ (иногда записывается как $7 \equiv 2 \pmod 5$). Данный пример можно прочитать как «7 сравнимо с 2 по модулю 5». Аналогично:

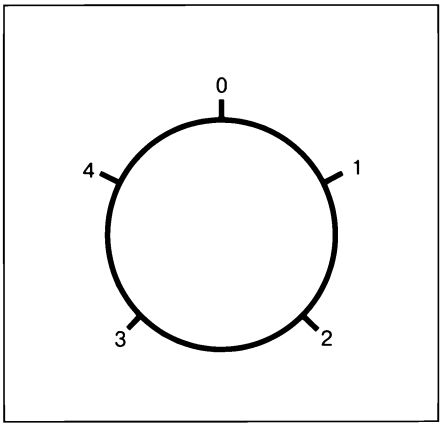
$$8 = 1 \bmod 7;$$
$$54 = 2 \bmod 13;$$
$$170 = 0 \bmod 17$$

и так далее.

Классическим способом представления такой концепции являются часы, изображенные на рис. 5.5.



Обычные часы заново начинают отсчет на отметке 12: 12 — это 0, 13 — 1, 14 — 2 и т. д.



Целые числа по модулю 5: 5 — это 0, 6 — 1 и т. д.

Рис. 5.5. Группу целых чисел по модулю простого числа 5 можно представить в виде часов, которые сбрасываются на 0 после числа 4. Таким образом, 5 принимает значение 0, 6 — 1, 7 — 2, 8 — 3, 9 — 4, 10 — 0 и т. д.

На таком множестве чисел вполне естественно определить модульное умножение. В качестве примера рассмотрим следующее произведение:

$$3 \times 2 = 6.$$

Основываясь на полученных только что знаниях, мы можем сказать, что 6 сравнимо с 1 по модулю 5, следовательно, данный пример можно переписать так:

$$3 \times 2 = 1 \bmod 5.$$

Довольно просто, не правда ли? Обратите внимание на то, что 3 здесь является обратной величиной по отношению к 2 и наоборот. Таким образом, возможна следующая запись:

$$3^{-1} = 2 \bmod 5.$$

Как правило, если контекст понятен, то часть с модулем (здесь mod 5) опускается. Поэтому не удивляйтесь, если не увидите ее в некоторых примерах, приведенных в этой книге.

ПРИМЕЧАНИЕ

Так получилось, что при использовании положительных чисел по модулю простого числа только *нулевой* элемент не имеет обратной величины. (Действительно, можно ли найти такой элемент b, при котором $0 \times b = 1 \bmod 5$?) Именно по этой причине ноль не является элементом группы.

Итак, у нас есть группа, которая включает в себя множество строго положительных целых чисел 1, 2... $p - 1$, где p — это простое число, и модульное умножение. Сформированная нами группа также имеет:

- *коммутативное свойство* — порядок операций в группе не имеет значения. Например, если a и b — два элемента группы, то $ab = ba$. Группа, обладающая этим свойством, часто называется *группой Галуа*;
- *конечное поле* — группа Галуа, которая имеет больше свойств, а также дополнительную операцию (в своем примере мы можем еще и складывать числа).

В связи с последним пунктом алгоритм Диффи — Хеллмана, определенный на группе такого типа, иногда называют *алгоритмом Диффи — Хеллмана в конечном поле* (Finite Field Diffie — Hellman, FFHD). Если вы поняли, что такое группа (а вы должны убедиться в этом, прежде чем продолжить), то перейдем к понятию *подгруппы*. Подгруппа — это просто группа внутри исходной группы. Другими словами, это подмножество элементов группы. Результатом операции над элементами подгруппы является другой элемент этой подгруппы, а каждый элемент подгруппы имеет в ней обратную величину.

Циклическая подгруппа — это подгруппа, которая может быть порождена от одного *образующего элемента*, или *основания*. Образующий элемент порождает циклическую подгруппу путем многократного умножения на самого себя. Например, образующий элемент 4 определяет подгруппу, состоящую из чисел 1 и 4:

$$4 \bmod 5 = 4;$$

$$4 \times 4 \bmod 5 = 1;$$

$$4 \times 4 \times 4 \bmod 5 = 4 \text{ (возвращаемся к началу)};$$

$$4 \times 4 \times 4 \times 4 \bmod 5 = 1$$

и так далее.

ПРИМЕЧАНИЕ

$4 \times 4 \times 4$ можно записать как 4^3.

Бывает так, что, когда модуль — это простое число, каждый элемент группы является образующим элементом подгруппы. Эти подгруппы могут иметь различные размеры, которые называются *порядками* (рис. 5.6).



Рис. 5.6. Различные подгруппы мультипликативной группы по модулю 5. Все они включают число 1 (нейтральный элемент) и имеют различные порядки (количество элементов)

Итак, на данный момент вам должно быть понятно следующее.

- Группа — это множество чисел с бинарной операцией, отвечающее некоторым свойствам (замкнутость, ассоциативность, наличие нейтрального и обратного элементов).
- Алгоритм Диффи — Хеллмана работает в группе Галуа (группе с коммутативностью), образованной множеством строго положительных чисел вплоть до простого числа (не включая его) и модульным умножением.
- В группе Диффи — Хеллмана каждый элемент является образующим элементом подгруппы.

Группы являются центром огромного количества различных криптографических примитивов. Важно иметь хорошее представление о теории групп, чтобы понимать, как работают другие криптографические примитивы.

5.2.2. Задача дискретного логарифма — основа алгоритма Диффи — Хеллмана

Криптостойкость алгоритма обмена ключами Диффи — Хеллмана основывается на трудноразрешимой *задаче дискретного логарифма* в группе, о которой я кратко расскажу в этом разделе.

Представьте, что я беру образующий элемент, например 3, и даю вам случайный элемент из тех, которые он может породить, скажем $2 = 3^x \bmod 5$ для некоторого неизвестного x. Спросить вас: «Что такое x?» — все равно что попросить найти дискретный логарифм 2 по основанию 3. Таким образом, задача дискретного логарифма в группе сводится к выяснению того, сколько раз был умножен образующий элемент на самого себя, чтобы получился заданный элемент группы. Это очень важная часть! Потратьте несколько минут на ее осмысление, прежде чем продолжить.

В данном примере можно быстро определить, что ответом является 3 (действительно, $3^3 = 2 \bmod 5$). Но если мы выберем простое число намного больше 5, то найти решение станет значительно сложнее. В этом и заключается секрет алгоритма Диффи — Хеллмана. Теперь вы знаете достаточно для того, чтобы понять, как сгенерировать пару ключей.

1. Все участники должны договориться о большом простом числе p и образующем элементе g.
2. Каждый участник генерирует случайное число x, которое становится его закрытым ключом.
3. Каждый участник получает свой открытый ключ в виде $g^x \bmod p$.

То, что задача дискретного логарифма *трудноразрешима*, означает, что никто не сможет восстановить закрытый ключ на основе открытого (рис. 5.7).



Рис. 5.7. Выбор закрытого ключа в алгоритме Диффи — Хеллмана схож с выбором индекса в списке чисел, сформированного образующим элементом g. Задача дискретного логарифма состоит в том, чтобы найти индекс из одного только числа

Практика показала, что имеющиеся алгоритмы для вычисления дискретных логарифмов неэффективны. Но если я предоставлю вам x, то в вашем распоряжении окажутся чрезвычайно полезные алгоритмы, позволяющие проверить правильность решения $g^x \bmod p$. Если интересно, современный метод вычисления возведения в степень по модулю называется *бинарным алгоритмом возведения в степень*. Он эффективно вычисляет результат, проходя по x бит за битом.

ПРИМЕЧАНИЕ

Как и во всем, что касается криптографии, не исключено, что решение можно просто угадать. Однако достаточно длинные параметры (здесь большое простое число) могут свести шанс угадать верно к минимуму. Даже после сотен лет случайных попыток вероятность найти решение будет статистически близка к нулю.

Отлично. Как же использовать всю эту математику в алгоритме обмена ключами Диффи — Хеллмана? Представим следующее:

- у Алисы есть закрытый ключ a и открытый ключ $A = g^a \bmod p$;
- у Боба есть закрытый ключ b и открытый ключ $B = g^b \bmod p$.

Зная открытый ключ Боба, Алиса может вычислить общий секрет как $B^a \bmod p$. Боб может произвести аналогичные действия посредством открытого ключа Алисы и своего закрытого ключа $A^b \bmod p$. Естественно, в результате этих двух вычислений получается одно и то же число:

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \bmod p.$$

В этом и заключается магия алгоритма Диффи — Хеллмана. С точки зрения стороннего наблюдателя, сами по себе открытые ключи A и B никак не помогут вычислить результат обмена ключами $g^{ab} \bmod p$. Далее вы узнаете, как этот алгоритм используется в реальных приложениях и какие стандарты его описывают.

ВЫЧИСЛИТЕЛЬНАЯ И РАСПОЗНАВАТЕЛЬНАЯ ЗАДАЧИ ДИФФИ — ХЕЛЛМАНА

Кстати, в теоретической криптографии то, что знание $g^a \bmod p$ и $g^b \bmod p$ не помогает вычислить $g^{ab} \bmod p$, называется *вычислительной задачей Диффи — Хеллмана* (Computational Diffie — Hellman Assumption, CDH). Ее часто путают с более строгой *распознавательной задачей Диффи — Хеллмана* (Decisional Diffie — Hellman Assumption, DDH), из названия которой следует, что при наличии $g^a \bmod p$, $g^b \bmod p$ и $z \bmod p$ никто не сможет с уверенностью сказать, является ли последний элемент результатом обмена ключами между двумя открытыми ключами ($g^{ab} \bmod p$) или просто случайным элементом группы. Оба эти условия являются полезными теоретическими предположениями, на основе которых построено множество различных криптографических алгоритмов.

5.2.3. Стандарты, описывающие алгоритм Диффи — Хеллмана

Теперь, зная принцип работы алгоритма Диффи — Хеллмана, можно понять, что стороны соединения должны согласовать набор параметров, в частности простое число p и образующий элемент g. Далее вы узнаете, как происходит выбор этих параметров в реальности и какие стандарты описывают алгоритм.

Сначала рассмотрим простое число p. Как я уже говорил, чем оно больше, тем лучше. Поскольку алгоритм Диффи — Хеллмана основан на задаче дискретного логарифма, его криптостойкость напрямую зависит от успешности известных решений, найденных для данной задачи. Любые достижения в этой области могут ослабить алгоритм. Со временем нам удалось получить достаточно хорошее представление о том, насколько быстро или медленно идет прогресс в этом направлении

и какова должна быть защита. В настоящее время известно, что лучше всего использовать простое число длиной 2048 бит.

ПРИМЕЧАНИЕ

На сайте https://keylength.com размещены общие рекомендации для распространенных криптографических алгоритмов по выбору длины параметров. Они были сформулированы на основании авторитетных документов, подготовленных исследовательскими группами и такими государственными органами, как ANSSI (Франция), NIST (США) и BSI (Германия). Несмотря на то что эти рекомендации могут идти вразрез друг с другом, часто они определяют одинаковый порядок величин.

В прошлом многие библиотеки и программы сами генерировали и жестко кодировали свои параметры. К сожалению, иногда это приводило либо к неэффективности, либо, что еще хуже, к полному взлому таких параметров. В 2016 году выяснилось, что годом ранее в популярном инструменте командной строки Socat используемая по умолчанию группа Диффи — Хеллмана была заменена ее уязвимым вариантом. Тогда возник вопрос: было ли это ошибкой или намеренно созданным черным ходом? Может показаться, что более удачным решением в данной ситуации было бы применение стандартизированных групп Диффи — Хеллмана, но это не так. Спустя всего несколько месяцев после обнародования проблемы Socat Антонио Сансо, изучая стандарт RFC 5114, обнаружил, что в нем указаны и уязвимые группы Диффи — Хеллмана.

В связи со всеми этими проблемами новые протоколы и библиотеки стали либо отказываться от алгоритма Диффи — Хеллмана в пользу его вариации на эллиптических кривых (Elliptic curve Diffie — Hellman, ECDH), либо задействовать группы из улучшенного стандарта RFC 7919 (https://www.rfceditor.org/info/ rfc7919). В настоящее время RFC 7919, в котором определены несколько групп различного размера и уровня защиты, считается лучшей практикой. Например, одной из таких групп является ffdhe2048. Она определена 2048-битным модулем простого числа p = 32 317 006 071 311 007 300 153 513 477 825 163 362 488 057 133 489 075 174 588 434 139 269 806 834 136 210 002 792 056 362 640 164 685 458 556 357 935 330 816 928 829 023 080 573 472 625 273 554 742 461 245 741 026 202 527 916 572 972 862 706 300 325 263 428 213 145 766 931 414 223 654 220 941 111 348 629 991 657 478 268 034 230 553 086 349 050 635 557 712 219 187 890 332 729 569 696 129 743 856 241 741 236 237 225 197 346 402 691 855 797 767 976 823 014 625 397 933 058 015 226 858 730 761 197 532 436 467 475 855 460 715 043 896 844 940 366 130 497 697 812 854 295 958 659 597 567 051 283 852 132 784 468 522 925 504 568 272 879 113 720 098 931 873 959 143 374 175 837 826 000 278 034 973 198 552 060 607 533 234 122 603 254 684 088 120 031 105 907 484 281 003 994 966 956 119 696 956 248 629 032 338 072 839 127 039 и образующим элементом g = 2.

ПРИМЕЧАНИЕ

Как правило, в качестве образующего элемента выбирается число 2, потому что компьютеры довольно эффективно выполняют умножение на 2 с помощью простой инструкции сдвига влево (<<).

Ко всему прочему, размер группы, или *порядок*, задается как $q = (p - 1) / 2$. Значит, длина как закрытого, так и открытого ключа будет составлять около 2048 бит, что на практике считается довольно большим размером (сравните, например, с симметричными ключами, обычная длина которых 128 бит). Как будет показано в следующем разделе, определение группы на эллиптических кривых позволяет получить ключи гораздо меньшего размера при той же степени защиты.

5.3. АЛГОРИТМ ОБМЕНА КЛЮЧАМИ ДИФФИ — ХЕЛЛМАНА НА ЭЛЛИПТИЧЕСКИХ КРИВЫХ

Оказывается, алгоритм Диффи — Хеллмана, о котором мы только что говорили, может быть реализован в различных типах групп, а не только в мультипликативных группах по модулю простого числа. Также группу можно составить из эллиптических кривых — одного из типа кривых, изучаемых в математике. Эту идею выдвинули параллельно Нил Коблиц и Виктор С. Миллер в 1985 году. Приняли ее значительно позже, в 2000-м, когда криптографические алгоритмы на основе эллиптических кривых стали входить в криптографические стандарты.

Эллиптическую криптографию приняли быстро, а все потому, что она, в отличие от предыдущего поколения криптографии с открытым ключом, предоставила ключи гораздо меньшего размера. По сравнению с рекомендованными в алгоритме Диффи — Хеллмана 2048-битными параметрами в варианте алгоритма на эллиптических кривых возможны параметры размером всего 256 бит.

5.3.1. Что такое эллиптические кривые

Теперь объясним, как работают эллиптические кривые. Прежде всего необходимо понимать, что эллиптические кривые — это просто кривые! То есть они определяются всеми координатами x и y, являющимися решениями уравнения. Если говорить конкретнее, это уравнение:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

для некоторых a_1, a_2, a_3, a_4 и a_6. Отмечу, что для большинства кривых, применяемых в наши дни на практике, это уравнение можно упростить до *краткой формы Вейерштрасса*:

$$y^2 = x^3 + ax + b, \text{ где } 4a^3 + 27b^2 \neq 0.$$

Лишь два типа кривых, *бинарные кривые* и *кривые над полями характеристики 3*, не допускают данного упрощения, но они используются довольно редко, поэтому в оставшейся части главы будем применять краткую форму Вейерштрасса. На рис. 5.8 показан пример эллиптической кривой с двумя произвольно взятыми точками.

В какой-то момент существования эллиптических кривых выяснилось, что над ними можно построить *группу*. Таким образом, реализация алгоритма Диффи — Хеллмана на таких группах не составила труда. В этом разделе я дам

наглядное представление об эллиптической криптографии.

Группы над эллиптическими кривыми часто определяются как *аддитивные*. В отличие от мультипликативных групп из предыдущего раздела в них используется знак +.

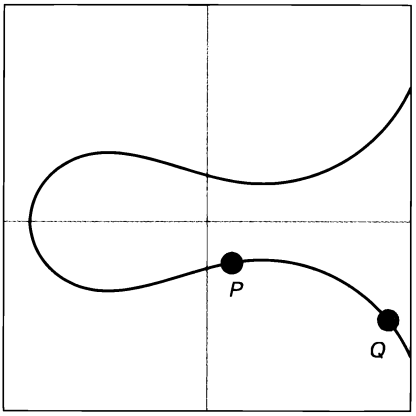


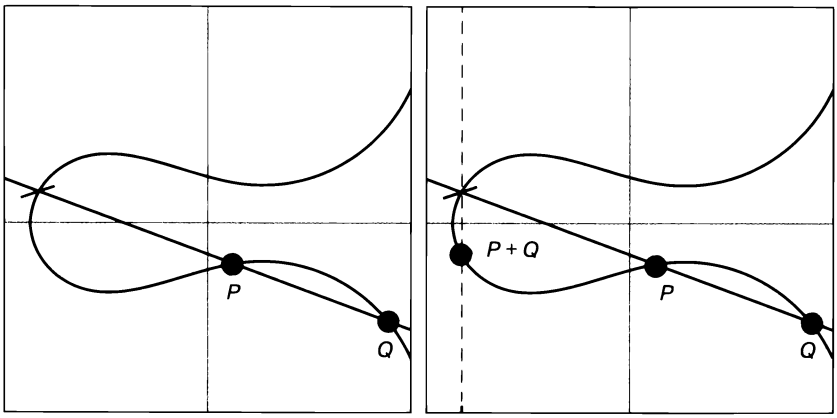
Рис. 5.8. Пример эллиптической кривой, определенной уравнением

ПРИМЕЧАНИЕ

Использование сложения или умножения не имеет большого значения на практике — это просто вопрос предпочтения. По большей части криптография задействует мультипликативную форму записи. Однако в литературе, посвященной эллиптическим кривым, принято задействовать аддитивную систему обозначений, поэтому я буду применять именно ее, когда буду говорить о группах на эллиптических кривых.

На этот раз начнем с определения операции, а не элементов группы. *Операция сложения* выполняется следующим образом (рис. 5.9).

1. Через две точки, которые нужно добавить, проводится прямая. Она пересекает кривую в третьей точке.
2. Через новую точку проводится вертикальная линия. Она пересекает кривую еще в одной точке.
3. Эта точка — результат сложения двух исходных точек.



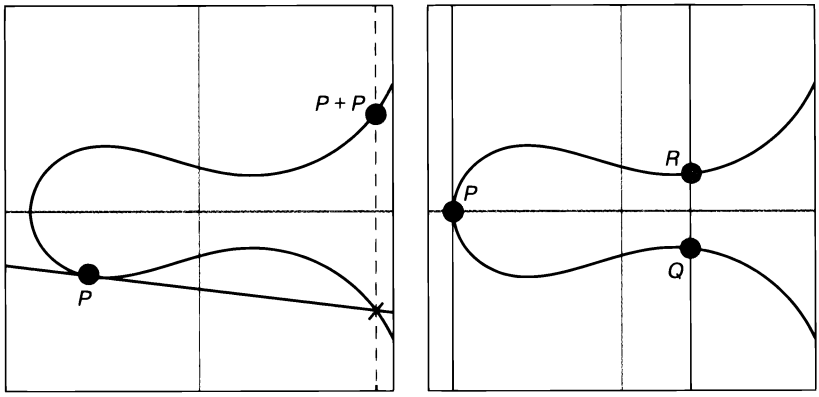
1. Линия, проведенная через точки *P* и *Q*, пересекает кривую в третьей точке

2. Вертикальная линия, проведенная через третью точку, пересекает кривую в точке *P* + *Q*

Рис. 5.9. Операция сложения может быть геометрически определена над точками эллиптической кривой

Рассмотрим два особых случая, когда это правило не работает.

- *Как сложить точку саму с собой?* Ответ: провести касательную к этой точке (вместо того чтобы проводить прямую между двумя точками).
- *Что произойдет, если линия, проведенная на шаге 1 или шаге 2, не пересечет кривую ни в какой другой точке?* Это усложняет дело, тем не менее нам нужно, чтобы этот особый случай работал и давал результат. Решением здесь является определение выдуманной точки (того, что мы сами придумали). Такая точка называется *бесконечно удаленной точкой* (как правило, она обозначается прописной буквой *O*). Эти два частных случая наглядно продемонстрированы на рис. 5.10.



Чтобы сложить точку саму с собой, нужно провести касательную в ней и выполнить остальную часть операции

Иногда сложение точек приводит к появлению бесконечно удаленной точки: $P + P = O$ и $R + Q = O$

Рис. 5.10. Продолжение рис. 5.9. Определение сложения на эллиптической кривой происходит и когда точка складывается сама с собой или когда в результате взаимоисключения двух точек получается бесконечно удаленная точка O

Я знаю, бесконечно удаленная точка — странность нового уровня, но не стоит сильно переживать по этому поводу. На самом деле эту точку мы придумываем специально для того, чтобы операция сложения работала. Кстати, она ведет себя как ноль и, таким образом, является нейтральным элементом:

$$O + O = O,$$

и для каждой точки P на кривой:

$$P + O = P.$$

Все хорошо. На данный момент мы знаем, что для создания группы над эллиптической кривой необходимы:

- уравнение эллиптической кривой, определяющее множество действительных точек;

- определение того, что означает сложение в этом множестве;
- придуманная точка, называемая бесконечно удаленной точкой.

Я понимаю, что информации много, но необходимо обратить внимание еще на один, последний, момент. Эллиптическая криптография использует рассмотренный ранее тип группы, определенный над *конечным полем*. На практике координатами являются числа 1, 2... $p - 1$ для некоторого большого простого числа p. Это должно показаться знакомым! Таким образом, следует представлять эллиптическую криптографию в виде графика, который больше похож на тот, что приведен на рис. 5.11, *справа*.

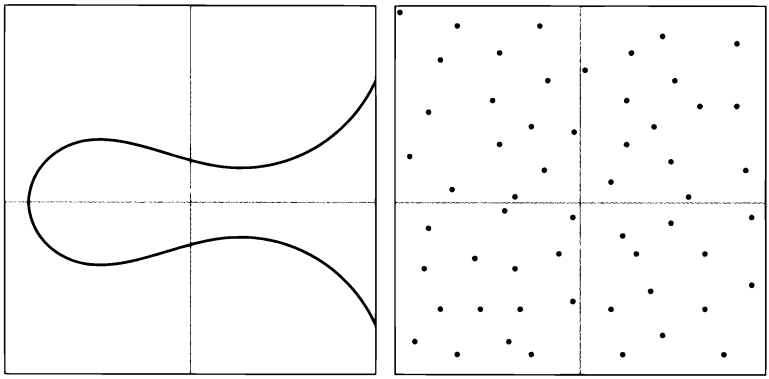


Рис. 5.11. На практике эллиптическая криптография (elliptic curve cryptography, ECC) в основном задается эллиптическими кривыми с координатами по модулю большого простого числа p. Следовательно, то, что мы применяем в криптографии, гораздо больше похоже на правый график, чем на левый

И все! Теперь у нас есть группа, на которой мы можем выполнять криптографические задачи, подобно тому как для алгоритма Диффи — Хеллмана имеем группу, состоящую из чисел (исключая 0) по модулю простого числа и операции умножения. Как можно использовать группу Диффи — Хеллмана, определенную на эллиптических кривых? Чтобы ответить на вопрос, посмотрим, как в этой группе работает *дискретный логарифм*.

Возьмем точку G и сложим ее с собой x раз, чтобы получить другую точку P посредством определенной нами операции сложения. Мы можем записать эту операцию как $P = G + ... + G$ (x раз) или использовать некоторые изыски математического синтаксиса и ограничиться выражением $P = [x]G$, что означает G, умноженное на x. *Задача дискретного логарифма на эллиптической кривой* (Elliptic Curve Discrete Logarithm Problem, ECDLP) состоит в том, чтобы найти число x, зная только P и G.

ПРИМЕЧАНИЕ

Мы называем $[x]G$ скалярным произведением, поскольку в таких группах x обычно называется *скаляром*.

5.3.2. Как работает протокол обмена ключами Диффи — Хеллмана на эллиптических кривых

Теперь, когда мы построили группу на эллиптических кривых, можем реализовать на ней алгоритм обмена ключами Диффи — Хеллмана (ECDH). Для генерации пары ключей в ECDH выполняется следующее.

1. Все участники согласовывают уравнение эллиптической кривой, конечное поле (скорее всего, простое число) и образующий элемент G (в криптографии с эллиптическими кривыми его обычно называют *базовой точкой*).
2. Каждый участник генерирует случайное число x, которое становится его закрытым ключом.
3. Каждый участник получает свой открытый ключ в виде $[x]G$.

Поскольку задача дискретного логарифма на эллиптической кривой трудноразрешима, то, как вы уже могли догадаться, никому не удастся восстановить закрытый ключ, просто перехватив открытый (рис. 5.12).

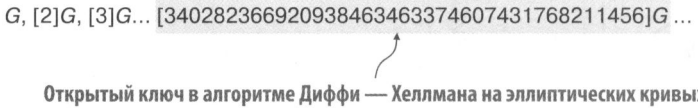


Рис. 5.12. В ECDH выбор закрытого ключа схож с выбором индекса в списке чисел, сформированного образующим элементом (или базовой точкой) G. Задача дискретного логарифма на эллиптической кривой (ECDLP) заключается в том, чтобы найти индекс из одного только числа

Все это может показаться несколько запутанным, поскольку операция, которую мы определили для группы Диффи — Хеллмана, была умножением, а для эллиптической кривой использовано сложение. И вновь это различие не имеет никакого значения, поскольку операции эквивалентны. На рис. 5.13 можно сравнить оба варианта.



Рис. 5.13. Сравнение группы, используемой в алгоритме Диффи — Хеллмана, с группой, применяемой в алгоритме Диффи — Хеллмана на эллиптических кривых

Теперь не должно остаться сомнений в том, что для криптографии важно только наличие группы с определенной на ней операцией и трудность задачи дискретного логарифма для этой группы. Для наглядности на рис. 5.14 показано различие задач дискретного логарифма для двух рассмотренных типов групп.



Дискретный логарифм в алгоритме Диффи — Хеллмана



Дискретный логарифм в алгоритме Диффи — Хеллмана на эллиптических кривых

Рис. 5.14. Сравнение задачи дискретного логарифма по модулю больших простых чисел и задачи дискретного логарифма в эллиптической криптографии. Обе они относятся к алгоритму обмена ключами Диффи — Хеллмана, поскольку основная задача заключается в нахождении закрытого ключа через открытый

И последнее замечание по теории: группа, сформированная над эллиптическими кривыми, отличается от группы, сформированной на основе строго положительных целых чисел по модулю простого числа. Некоторые из этих различий обусловливают хорошую защиту групп на эллиптических кривых от самых мощных из известных атак на алгоритм Диффи — Хеллмана — *атаки исчисления порядка* или *решета числового поля*. Это основная причина того, почему параметры ECDH могут быть значительно короче параметров Диффи — Хеллмана при обеспечении одинакового уровня защиты.

Итак, с теорией покончено. Вернемся к определению ECDH. Представьте:

- у Алисы есть закрытый ключ a и открытый ключ $A = [a]G$;
- у Боба есть закрытый ключ b и открытый ключ $B = [b]G$.

С помощью открытого ключа Боба Алиса может вычислить общий секрет $[a]B$. Боб может произвести аналогичные вычисления с открытым ключом Алисы и своим закрытым ключом: $[b]A$. Естественно, мы видим, что результат этих двух вычислений получается одинаковым:

$$[a]B = [a][b]G = [ab]G = [b][a]G = [b]A.$$

Ни один пассивный злоумышленник не сможет получить общий секрет, просто наблюдая за открытыми ключами. Звучит знакомо, правда? Далее поговорим о стандартах.

5.3.3. Стандарты, описывающие алгоритм Диффи — Хеллмана на эллиптических кривых

> Эллиптическая криптография не утратила своей актуальности с момента ее первого представления в 1985 году <...> США, Великобритания, Канада и некоторые другие страны НАТО задействовали ту или иную форму эллиптической криптографии при разработке будущих систем защиты внутренней и внешней секретной правительственной информации.
>
> *АНБ (The Case for Elliptic Curve Cryptography, 2005)*

Стандартизация ECDH проходила довольно хаотично. Многие органы по стандартизации работали над определением большого количества различных кривых, что привело к вспышке множества пламенных споров по поводу того, какая кривая более безопасна или более эффективна. Большое количество исследований, проведенных в основном под руководством Дэниела Дж. Бернштейна, указывало на то, что ряд кривых, стандартизированных NIST, потенциально могли входить в более слабый класс кривых, известных только АНБ.

> Я больше не доверяю константам. Я считаю, что АНБ манипулирует ими, используя свое влияние на сферу криптографии.
>
> *Брюс Шнайер (The NSA Is Breaking Most Encryption on the Internet, 2013)*

В настоящее время большинство используемых кривых регламентировано всего в паре стандартов, и чаще всего приложения применяют две кривые: P-256 и Curve25519. О них и пойдет речь в оставшейся части раздела.

NIST FIPS 186-4. Digital Signature Standard («Стандарт цифровой подписи»), первоначально опубликованный как стандарт для подписей в 2000 году, содержит приложение с определением 15 кривых для использования в ECDH. Одна из них, P-256, наиболее широко применяется в Интернете. Она также была определена в стандартах SEC (Standards for Efficient Cryptography) 2, v2 Recommended Elliptic Curve Domain Parameter («Рекомендуемые домен-параметры для эллиптических кривых»), опубликованных в 2010 году, но под другим названием — secp256r1. Кривая P-256 определяется с помощью краткой формы уравнения Вейерштрасса:

$$y^2 = x^3 + ax + b \bmod p,$$

где $a = -3$, b = 41 058 363 725 152 142 129 326 129 780 047 268 409 114 441 015 993 725 554 835 256 314 039 467 401 291 и

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1.$$

Оно определяет кривую простого порядка n = 115 792 089 210 356 248 762 697 446 949 407 573 529 996 955 224 135 760 342 422 259 061 068 512 044 369, то есть на кривой имеется ровно n точек, включая бесконечно удаленную точку. Базовая точка задается здесь как G = (48 439 561 293 906 451 759 052 585 252 797 914 202

762 949 526 041 747 995 844 080 717 082 404 635 286, 36 134 250 956 749 795 798 585 127 919 587 881 956 611 106 672 985 015 071 877 198 253 568 414 405 109).

Кривая P-256 предоставляет 128 бит защиты. Для приложений, использующих другие криптографические алгоритмы, обеспечивающие 256 бит защиты (например, AES с 256-битным ключом), в том же стандарте указана кривая P-521.

МОЖНО ЛИ ДОВЕРЯТЬ КРИВОЙ P-256?

Интересно, что P-256 и другие кривые, определенные в FIPS 186-4, как утверждается, генерируются из *зерна* (seed). В качестве него для P-256 используется байтовая строка 0xc49d360886e704936a6678e1139d26b7819f7e90.

Ранее мы уже касались значения nothing-up-my-sleeve — константы, призванной доказать, что при разработке алгоритма не была создана возможность для появления черного хода. К сожалению, нет особых пояснений относительно зерна P-256, кроме информации о том, что оно задается по параметру кривой.

В документе RFC 7748. *Elliptic Curves for Security* («Эллиптические кривые для обеспечения криптостойкости»), опубликованном в 2016 году, определены две кривые: Curve25519 и Curve448. Curve25519 обеспечивает примерно 128 бит защиты, а Curve448 — около 224 бит защиты. Они применяются в протоколах в качестве подстраховки на случай возможного прогресса в области атак на эллиптические кривые. Здесь я буду говорить только о кривой Curve25519, которая представляет собой кривую Монтгомери, определенную уравнением:

$$y^2 = x^3 + 486\,662x^2 + x \bmod p,$$

где $p = 2^{255} - 19$.

Порядком Curve25519 является $n = 2^{252} + 27\,742\,317\,777\,372\,353\,535\,851\,937\,790\,883\,648\,493$.

В качестве базовой точки используется G = (9, 14 781 619 447 589 544 791 020 593 568 409 986 887 264 606 134 616 475 288 964 881 837 755 586 237 401).

Комбинацию ECDH и Curve25519 часто называют *X25519*.

5.4. АТАКИ ЧЕРЕЗ МАЛЫЕ ПОДГРУППЫ И ДРУГИЕ АСПЕКТЫ БЕЗОПАСНОСТИ

Сегодня *я бы посоветовал вместо алгоритма Диффи — Хеллмана задействовать его вариацию на эллиптических кривых*. В пользу ECDH говорят оптимальный размер ключей, отсутствие известных мощных атак против него, качество доступных реализаций, а также фиксированность и хорошая стандартизация эллиптических кривых, в отличие от беспорядочных групп Диффи — Хеллмана. Последнее замечание очень важно! Алгоритм Диффи — Хеллмана подразумевает под собой

возможность применения скомпрометированных стандартов, например RFC 5114, слишком расплывчатых протоколов (многие протоколы, такие как старые версии TLS, не указывают, какие группы Диффи — Хеллмана следует брать), программного обеспечения с уязвимыми пользовательскими группами Диффи — Хеллмана (проблема утилиты Socat, о которой говорилось ранее) и т. д.

Если вам все же приходится использовать алгоритм Диффи — Хеллмана, обязательно *придерживайтесь актуальных стандартов*. В упомянутых ранее стандартах в качестве модуля применяются безопасные простые числа — простые числа вида $p = 2q + 1$, где q — другое простое число. Дело в том, что группы такого вида имеют только две подгруппы: малую размером 2 (порожденную -1) и большую размером q. (Это, кстати, лучшее, что можно получить, так как групп простого порядка в алгоритме Диффи — Хеллмана не существует.) Ограниченность малых подгрупп предотвращает тип *атаки через малые подгруппы* (подробнее об этом — чуть позже). Безопасные простые числа создают безопасные группы благодаря двум факторам.

- Порядок мультипликативной группы по модулю простого числа p вычисляется как $p - 1$.
- Порядок подгрупп группы — это множители порядка группы (теорема Лагранжа).

Следовательно, порядок нашей мультипликативной группы по модулю безопасного простого числа равен $p - 1 = (2q + 1) - 1 = 2q$ с коэффициентами 2 и q, а значит, в нее могут входить только подгруппы порядка 2 или q. В таких группах атаки через малые подгруппы невозможны, поскольку нет достаточного количества малых подгрупп. *Атака через малые подгруппы* — это атака на обмен ключами, при которой злоумышленник посылает несколько недействительных открытых ключей для постепенного раскрытия битов закрытого ключа жертвы, причем недействительные открытые ключи являются образующими элементами малых подгрупп.

Например, злоумышленник может выбрать -1 в качестве открытого ключа (образующий элемент подгруппы размером 2) и отправить его жертве. В результате обмена ключами получается общий секрет, который является элементом малой подгруппы (-1 или 1), так как жертва возвела образующий элемент малой подгруппы (открытый ключ злоумышленника) в степень своего закрытого ключа. В зависимости от того, что жертва делает с этим общим секретом, злоумышленник может раскрыть некоторую информацию о ее закрытом ключе.

Если бы в нашем примере с вредоносным открытым ключом закрытый ключ жертвы был четным, то общий секрет был бы равен 1, а если нечетным, то -1. Таким образом, злоумышленник узнает один бит информации — младший бит закрытого ключа. Наличие большого количества подгрупп разного размера может привести к тому, что атакующий будет получать все больше информации о закрытом ключе, пока не восстановит его целиком. Данная проблема проиллюстрирована на рис. 5.15.

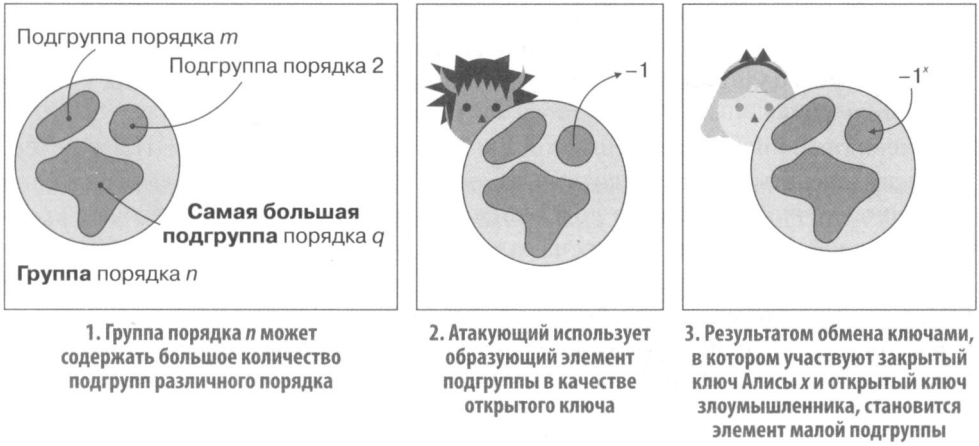


Рис. 5.15. Атака через малые подгруппы проводится на группах Диффи — Хеллмана с большим количеством подгрупп. Выбирая в качестве открытых ключей образующие элементы малых подгрупп, злоумышленник может постепенно раскрыть биты закрытого ключа жертвы

Всегда полезно проверять, входит ли полученный открытый ключ в правильную подгруппу, однако не во всех реализациях эта рекомендация выполняется. В 2016 году группа исследователей проанализировала 20 реализаций алгоритма Диффи — Хеллмана и обнаружила, что ни одна из них не проверяет открытые ключи (см. статью *Measuring small subgroup attacks against Diffie — Hellman* Л. Валента с соавторами). Убедитесь, что используемые вами реализации алгоритма Диффи — Хеллмана выполняют данную проверку! Это можно сделать, возведя открытый ключ в степень порядка подгруппы. Если он является элементом данной подгруппы, вернется нейтральный элемент.

В то же время эллиптические кривые позволяют создавать группы простого порядка. Они не имеют малых подгрупп (кроме подгруппы размера 1, порожденной нейтральным элементом), а значит, надежно защищены от атак через малые подгруппы. Но не все так радужно... В 2000 году Биль, Мейер и Мюллер обнаружили, что атаки через малые подгруппы возможны даже в таких группах простого порядка на эллиптических кривых. Это *атаки через недействительную кривую*.

Идея атаки через недействительную кривую заключается в следующем. Во-первых, формулы для реализации скалярного произведения для эллиптических кривых, использующих краткую форму Вейерштрасса $y^2 = x^3 + ax + b$ (например, P-256 от NIST), не зависят от переменной b. Следовательно, злоумышленник может найти другие кривые с аналогичным уравнением, в котором иным будет только значение b. Некоторые из этих кривых будут иметь большое количество малых подгрупп. Вы, наверное, догадываетесь, к чему я веду: злоумышленник выбирает точку на другой кривой с малыми подгруппами и отправляет ее на сервер, который

атакует. Сервер продолжает обмен ключами, выполняя скалярное произведение с данной точкой, а фактически обменивается ключами на другой кривой. В итоге этот трюк позволяет повторить атаку через малые подгруппы даже на кривых простого порядка.

Очевидный способ решения этой проблемы — проверка открытых ключей. Это легко сделать: нужно убедиться, что открытый ключ не является бесконечно удаленной точкой и что, если подставить координаты открытого ключа в уравнение кривой, они опишут точку на заданной кривой. К сожалению, в 2015 году Ягер, Швенк и Соморовский в работе *Practical Invalid Curve Attacks on TLS-ECDH* («Практические атаки на недействительные кривые TLS-ECDH») показали, что в некоторых популярных реализациях алгоритма эти проверки не выполняются. Я бы посоветовал в ходе работы с ECDH использовать алгоритм обмена ключами X25519 из-за хорошего качества его исходной конструкции, учитывающей атаки через недействительные кривые, качества доступных реализаций и устойчивости к атакам по времени за счет конструкции.

Кривая Curve25519 имеет один нюанс: она не является группой простого порядка и имеет малую подгруппу размером 8 и большую подгруппу для ECDH. Кроме того, ни в исходной конструкции, ни в библиотеках не была реализована проверка получаемых точек, что привело к появлению проблем в различных типах протоколов, использующих данный примитив более экзотическим образом. (Одна из таких проблем была обнаружена в протоколе обмена сообщениями Matrix, о котором будет рассказано в главе 11.)

Отсутствие проверки открытых ключей может привести к неожиданным вариантам поведения X25519. Причина заключается в том, что алгоритм обмена ключами *не учитывает взаимодействия*: он не позволяет обеим сторонам влиять на конечный результат обмена. В частности, один из участников может сделать результат обмена нулевым, послав в качестве открытого ключа точку в малой подгруппе. RFC 7748 упоминает эту проблему и предлагает проверять, что получившийся общий секрет не является нулевым, но при этом предоставляет исполнителю право решать, выполнять эту проверку или нет! Я бы рекомендовал убедиться в том, что ваша реализация ее выполняет, хотя маловероятно, что вы столкнетесь с какими-то проблемами, если только не используете X25519 каким-нибудь нестандартным образом.

Поскольку многие протоколы основываются на Curve25519, ее слабые места стали проблемой не только для обмена ключами. Решить данную проблему призван *Ristretto* — проект интернет-документа, который в скором времени станет RFC. Он представляет собой конструкцию, которая эффективно имитирует кривую простого порядка путем введения дополнительного уровня кодирования для Curve25519 (см. https://tools.ietf.org/html/drafthdevalencecfrgristretto01). Эта конструкция находит все более широкое применение, поскольку упрощает условия обеспечения безопасности, принимаемые другими типами криптографических примитивов, которые хотят воспользоваться преимуществами Curve25519, но нуждаются в поле простого порядка.

РЕЗЮМЕ

- Неаутентифицированный обмен ключами позволяет двум сторонам согласовать общий секрет без риска его перехвата пассивным злоумышленником-посредником (MITM).
- Аутентифицированный обмен ключами не позволяет активному посреднику выдать себя за одну из сторон соединения, а взаимно аутентифицированный обмен ключами — за обе стороны.
- Можно выполнить аутентифицированный обмен ключами, зная открытый ключ другой стороны, но этот подход не всегда работает. С помощью подписей можно получить доступ к более сложным сценариям шифрования (см. главу 7).
- Алгоритм Диффи — Хеллмана — первый изобретенный алгоритм обмена ключами, который до сих пор находит широкое применение.
- Рекомендуемым стандартом для работы с алгоритмом Диффи — Хеллмана является RFC 7919, который содержит несколько параметров на выбор. Наименьший вариант — это рекомендуемый 2048-битный параметр, состоящий из простого числа.
- Алгоритм Диффи — Хеллмана на эллиптических кривых (ECDH) позволяет работать с ключами гораздо меньшего размера. Для обеспечения 128-битной безопасности алгоритм Диффи — Хеллмана требует 2048-битных параметров, в то время как его вариация на эллиптических кривых — 256-битных.
- Наиболее широко распространенные кривые для ECDH — P-256 и Curve25519. Обе обеспечивают 128-битную защиту. Для обеспечения 256-разрядной защиты определены P-521 и Curve448.
- Реализации должны проверять получаемые открытые ключи, поскольку некорректные ключи становятся причиной многих ошибок.

6 Асимметричный и гибридный виды шифрования

В этой главе

- ✓ Асимметричное шифрование секретов с использованием открытого ключа.
- ✓ Гибридное шифрование данных с помощью открытого ключа.
- ✓ Стандарты асимметричного и гибридного видов шифрования.

В главе 4 вы узнали об аутентифицированном шифровании — криптографическом примитиве шифрования данных, ограниченном своей симметричностью (обе стороны соединения должны иметь один и тот же ключ). В этой главе я покажу, как обойти это ограничение, представив асимметричное шифрование — примитив, позволяющий шифровать с использованием чужого открытого ключа, не зная соответствующего ему закрытого. Для данного вида шифрования все еще необходима пара ключей, однако в самом процессе вместо симметричных ключей применяются открытые ключи.

В середине главы вы увидите, что асимметричное шифрование тоже имеет свои границы, связанные со скоростью шифрования и размером шифруемых данных. Я покажу, как устранить эту проблему с помощью так называемого гибридного шифрования — комбинации асимметричного и аутентифицированного видов шифрования. Приступим!

ПРИМЕЧАНИЕ

Чтобы хорошо понять эту главу, рекомендую прочитать главу 4, посвященную аутентифицированному шифрованию, и главу 5 об обмене ключами.

6.1. ЧТО ТАКОЕ АСИММЕТРИЧНОЕ ШИФРОВАНИЕ

При изучении шифрования сообщений в первую очередь следует разобраться в *асимметричном шифровании*, известном также как *шифрование с открытым ключом*. Этот раздел посвящен данному криптографическому примитиву и его свойствам. Рассмотрим сценарий из реальной жизни — *зашифрованные электронные письма*.

Вероятно, вам известно, что все электронные письма отправляются в открытом виде и их может прочитать любой человек, находящийся между отправителем и почтовым провайдером адресата. Как исправить эту уязвимость? Например, можно использовать криптографический примитив типа AES-GCM из главы 4, то есть определить общий симметричный секрет отдельно для каждого адресанта.

УПРАЖНЕНИЕ

Использовать один и тот же общий секрет для всех — плохое решение. Можете сказать почему?

Однако невозможно заранее предугадать отправителя, а генерация и обмен симметричными ключами будут становиться сложнее по мере увеличения количества участников. В таком случае на помощь приходит асимметричное шифрование: оно позволяет любому человеку, владеющему вашим открытым ключом, зашифровать адресованные вам сообщения. К тому же только вы можете расшифровать эти сообщения, поскольку только у вас есть соответствующий закрытый ключ. Вот так выглядит асимметричное шифрование (рис. 6.1).

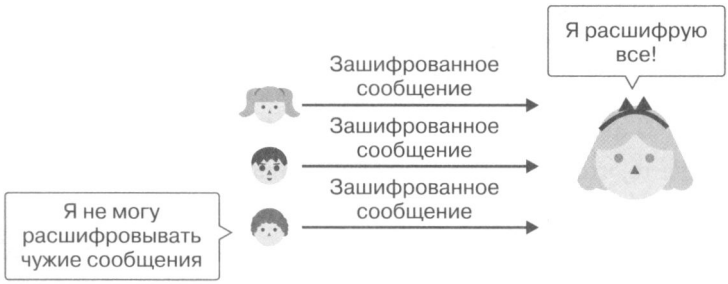


Рис. 6.1. При асимметричном шифровании любой человек может использовать открытый ключ Алисы для отправки ей зашифрованных сообщений. Расшифровать их может только Алиса, владеющая соответствующим закрытым ключом

Для настройки асимметричного шифрования сначала нужно с помощью какого-нибудь алгоритма сгенерировать пару ключей. Как и любая другая функция настройки криптографических алгоритмов, алгоритм генерации ключей принимает

параметр безопасности, который чаще всего понимается как вопрос «Какого размера должны быть ключи?», где «больше» значит «лучше» (рис. 6.2).

Алгоритм генерации ключей генерирует пару ключей, состоящую из двух частей: открытого ключа, который (как видно из названия) может находиться в открытом доступе и передаваться без особых проблем, и закрытого ключа, который должен оставаться в секрете. Как и в алгоритмах генерации ключей в других криптографических примитивах, для определения уровня битной криптостойкости алгоритма требуется параметр безопасности. Любой может использовать открытый ключ для шифрования сообщений, а закрытый ключ — для расшифровки (рис. 6.3). При этом аналогично аутентифицированному расшифрованию при нарушении целостности шифротекста расшифровка может завершиться неудачей.



Рис. 6.2. В начале асимметричного шифрования необходимо сгенерировать пару ключей. В зависимости от заданных параметров безопасности можно генерировать ключи различной степени защиты

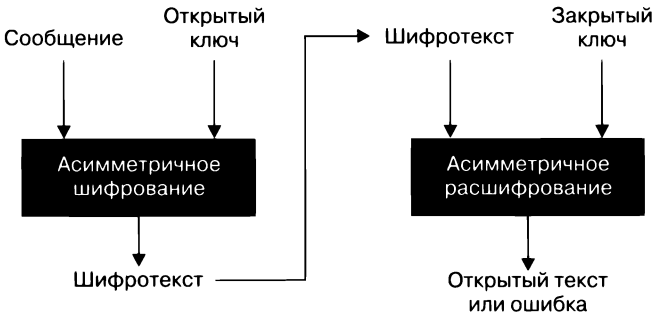


Рис. 6.3. Асимметричное шифрование позволяет зашифровать сообщение (открытый текст) с помощью открытого ключа получателя. Затем получатель может использовать другой алгоритм для расшифровки этого сообщения (шифротекста) с помощью соответствующего закрытого ключа

Обратите внимание на то, что мы до сих пор не упоминали аутентификацию. Рассмотрим обе стороны соединения.

- Вы шифруете сообщение, используя открытый ключ, который, как вы считаете, принадлежит Алисе.
- Алиса точно не знает, кто прислал ей сообщение.

Пока допустим, что мы получили открытый ключ Алисы из действительно надежного источника. В главе 7, посвященной цифровым подписям, вы узнаете, как реальные протоколы на практике решают данную проблему инициализации. Кроме того, в ней рассказано, как посредством криптографии можно сообщить Алисе, кто вы на самом деле. Спойлер: для этого нужно подписывать свои сообщения.

Перейдем к следующему разделу и посмотрим, как применяется асимметричное шифрование на практике и почему само по себе оно редко используется.

6.2. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ АСИММЕТРИЧНОГО ШИФРОВАНИЯ И ГИБРИДНОЕ ШИФРОВАНИЕ

Возможно, вы думаете, что для электронных писем достаточно асимметричного шифрования. На самом же деле возможности этого вида шифрования весьма ограничены из-за небольшой длины сообщений, которые оно может зашифровать. Кроме того, стоит отметить невысокую скорость асимметричного шифрования и расшифрования по сравнению с симметричным. Это связано с тем, что асимметричные конструкции реализуют математические операции, в отличие от симметричных примитивов, которые часто просто управляют битами.

Здесь будут рассмотрены эти ограничения и способы их преодоления, а также практические цели применения асимметричного шифрования. Данный раздел состоит из двух частей, посвященных двум основным случаям использования асимметричного шифрования.

- *Обмен ключами* — вы увидите, что примитив асимметричного шифрования вполне подходит для обмена ключами (или согласования ключей).
- *Гибридное шифрование* — вы увидите, что возможности применения асимметричного шифрования весьма ограничены в части размера шифруемого сообщения. Вы узнаете о более полезном примитиве для шифрования больших сообщений — гибридном шифровании.



Рис. 6.4. Для использования асимметричного шифрования в качестве примитива обмена ключами необходимо: *1* — сгенерировать симметричный ключ; *2* — зашифровать его с помощью открытого ключа Алисы

6.2.1. Обмен ключами и инкапсуляция ключа

Оказывается, асимметричное шифрование можно использовать для обмена ключами! Для этого можно сгенерировать симметричный ключ и зашифровать его с помощью открытого ключа Алисы, то есть *инкапсулировать этот ключ* (рис. 6.4).

Затем Алиса может расшифровать полученный шифротекст и таким образом увидеть симметричный ключ. Теперь и у нее, и у отправителя есть общий секрет! Весь процесс шифрования и расшифрования показан на рис. 6.5.

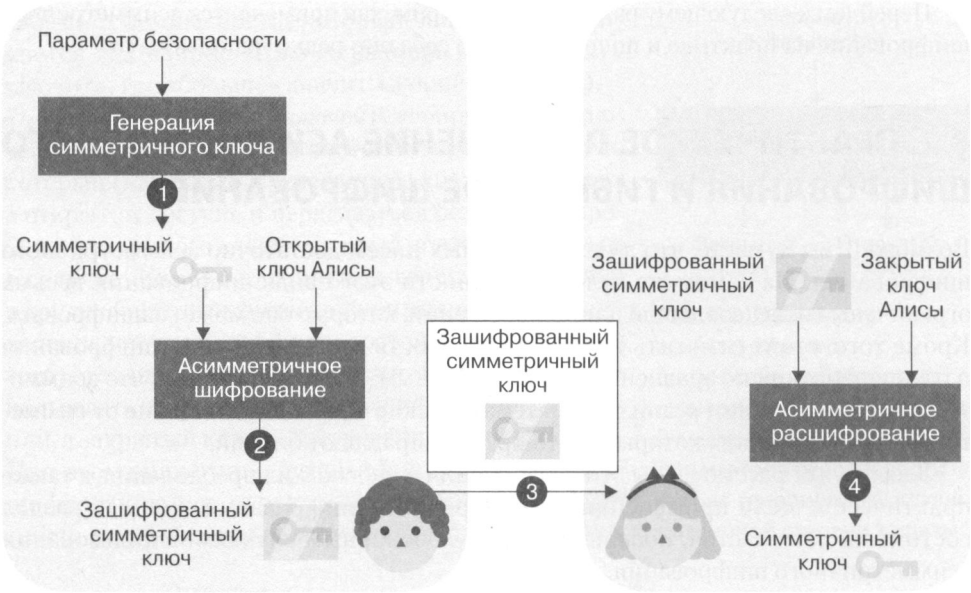


Рис. 6.5. Для использования асимметричного шифрования в качестве примитива обмена ключами можно сгенерировать симметричный ключ *(1)* и зашифровать его с помощью открытого ключа Алисы *(2)*. Получив этот ключ *(3)*, Алиса может расшифровать его с помощью соответствующего ему закрытого ключа *(4)*. В результате у Алисы и отправителя появится общий секрет, и никто другой не сможет заполучить его, зная только зашифрованный симметричный ключ

Асимметричное шифрование при обмене ключами обычно осуществляется с помощью алгоритма *RSA*, названного в честь его изобретателей Ривеста, Шамира и Адлемана. Сегодня RSA используется во многих интернет-протоколах. Тем не менее он потерял свой статус рекомендуемого способа обмена ключами, уступив его алгоритму Диффи – Хеллмана на эллиптических кривых (ECDH). Это обусловлено историческими причинами (в реализациях и стандартах RSA было обнаружено множество уязвимостей) и привлекательностью малых размеров параметров, предлагаемых ECDH.

6.2.2. Гибридное шифрование

Асимметричное шифрование позволяет шифровать сообщения только определенной длины. Так, размер открытых текстов, шифруемых с помощью RSA, ограничен параметрами безопасности, установленными при генерации пары ключей, а точнее, размером модуля. Сейчас при использовании параметров безопасности (модуля 4096 бит) предел составляет примерно 500 ASCII-символов, что довольно мало. Поэтому в большинстве приложений задействуется гибридное шифрование, пределы которого зависят от лимитов шифрования используемого алгоритма аутентифицированного шифрования.

Гибридное шифрование работает по той же схеме, что и асимметричное шифрование (рис. 6.6). Сообщения шифруются с помощью открытого ключа, а тот, кто владеет соответствующим закрытым ключом, может расшифровать их. Реальное различие этих двух видов шифрования заключается в предельных размерах шифруемых сообщений.

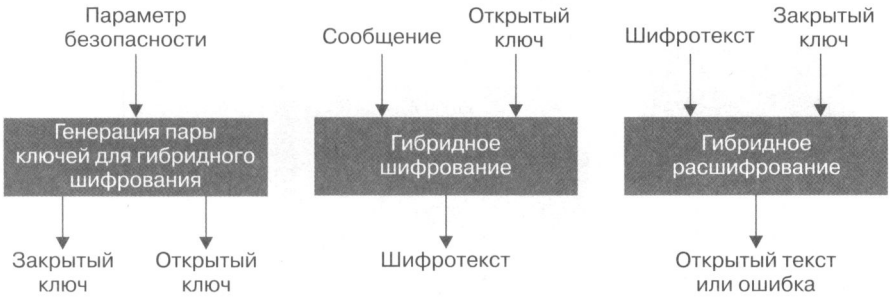


Рис. 6.6. Гибридное шифрование работает по той же схеме, что и асимметричное шифрование, за исключением того, что позволяет шифровать сообщения гораздо большего размера

Гибридное шифрование — это просто комбинация *асимметричного* и *симметричного* криптографических примитивов (отсюда и название). В частности, оно представляет собой неинтерактивный обмен ключами, за которым следует шифрование сообщения с помощью аутентифицированного алгоритма шифрования.

ВНИМАНИЕ

Вместо примитива аутентифицированного шифрования можно было бы использовать простой примитив симметричного шифрования. Однако такой метод не обеспечивает защиты от подделки зашифрованных сообщений. Именно поэтому на практике мы никогда не ограничиваемся только симметричным шифрованием (это было показано в главе 4).

Давайте узнаем, как работает гибридное шифрование. Как показано на рис. 6.7, сначала нужно сгенерировать симметричный ключ, а затем использовать его в алгоритме аутентифицированного шифрования для зашифровки сообщения Алисе.

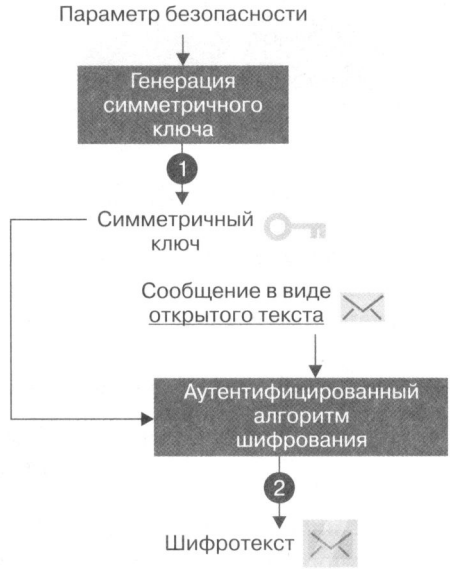


Рис. 6.7. Чтобы зашифровать сообщение для Алисы с помощью гибридного шифрования, необходимо: *1* — сгенерировать симметричный ключ для аутентифицированного алгоритма шифрования; *2* — использовать его для шифрования

Алиса не сможет расшифровать сообщение, не зная симметричного ключа. Как же передать его Алисе? Используя открытый ключ Алисы, асимметрично зашифровать симметричный ключ (рис. 6.8).

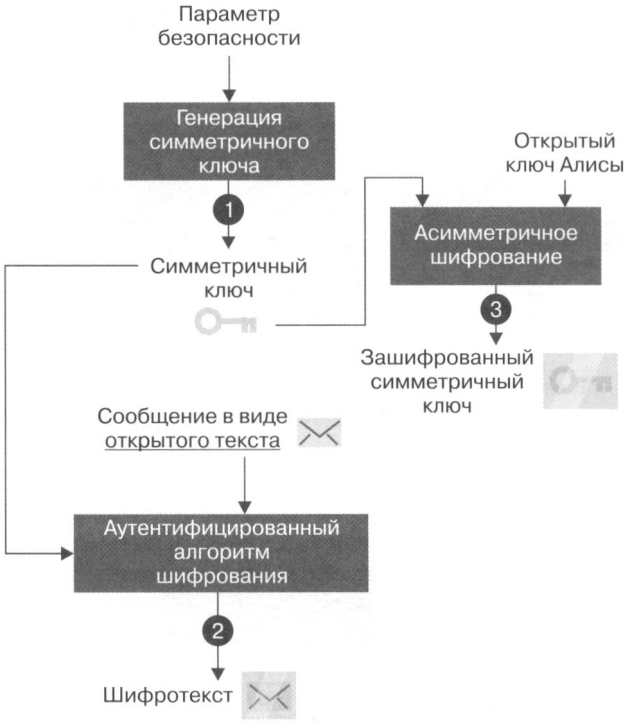


Рис. 6.8. Продолжение рис. 6.7: *3* — с помощью открытого ключа Алисы и алгоритма асимметричного шифрования шифруется сам симметричный ключ

В конечном итоге Алисе можно отправить оба результата шифрования:

- асимметрично зашифрованный симметричный ключ;
- симметрично зашифрованное сообщение.

Этой информации достаточно, чтобы Алиса смогла расшифровать сообщение. Полная последовательность действий показана на рис. 6.9.

Таким образом, мы можем взять лучшее из двух миров: сочетание асимметричного и симметричного шифрования позволяет шифровать большие объемы данных посредством открытого ключа. Асимметричную часть алгоритма мы часто называем *механизмом инкапсуляции ключа* (key encapsulation mechanism, KEM), симметричную часть — *механизмом инкапсуляции данных* (data encapsulation mechanism, DEM).

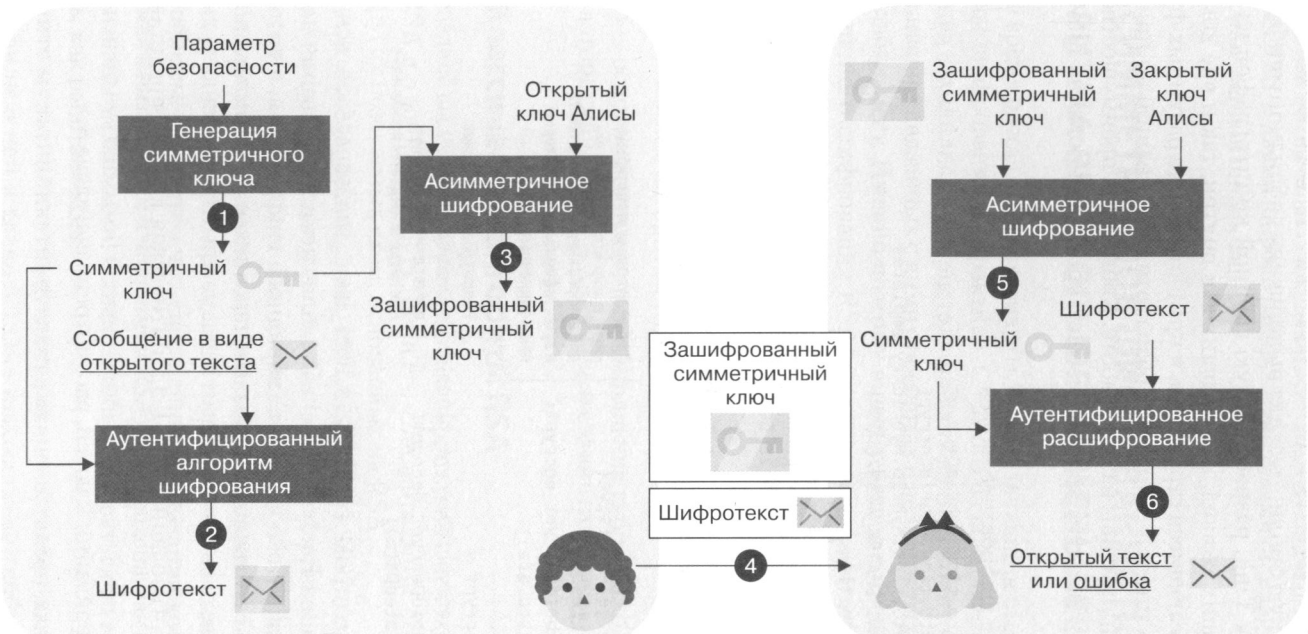


Рис. 6.9. Продолжение рис. 6.8: получив зашифрованный симметричный ключ и зашифрованное сообщение *(4)*, Алиса использует свой закрытый ключ для расшифровки симметричного ключа *(5)*, а симметричный ключ — для расшифровки сообщения *(6)*. (Обратите внимание на то, что шаги *5* и *6* могут завершиться неудачей и выдачей ошибки, если на шаге *4* в обмен данными вмешался посредник)

Прежде чем перейти к следующему разделу и познакомиться с различными алгоритмами и стандартами асимметричного и гибридного видов шифрования, давайте на практическом примере (листинг 6.1) посмотрим, как можно использовать криптографическую библиотеку для выполнения гибридного шифрования. Возьмем библиотеку Tink, разработанную группой криптографов из Google для поддержки больших команд, работающих как внутри, так и вне компании. Здравые решения, касающиеся структуры, и предоставление разумных функций в таком масштабном проекте помогают разработчикам правильно применять криптографические примитивы в своей работе. Кроме того, преимуществом Tink является ее доступность на нескольких языках программирования (Java, C++, ObjC и Golang).

Листинг 6.1. Гибридное шифрование на Java

```
import com.google.crypto.tink.HybridDecrypt;
import com.google.crypto.tink.HybridEncrypt;
import com.google.crypto.tink.hybrid.HybridKeyTemplates
➥ .ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM;
import com.google.crypto.tink.KeysetHandle;

KeysetHandle privkey = KeysetHandle.generateNew(
    ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM);
```

Генерация ключей для конкретной схемы гибридного шифрования

```
KeysetHandle publicKeysetHandle =
    privkey.getPublicKeysetHandle();
```

Получение открытого ключа, который можно передавать или выкладывать в открытый доступ

```
HybridEncrypt hybridEncrypt =
    publicKeysetHandle.getPrimitive(
        HybridEncrypt.class);
byte[] ciphertext = hybridEncrypt.encrypt(
    plaintext, associatedData);
```

Любой, кто знает этот открытый ключ, может использовать его для шифрования открытого текста и аутентифицировать некоторые присоединенные данные

```
HybridDecrypt hybridDecrypt =
    privkey.getPrimitive(HybridDecrypt.class);
byte[] plaintext = hybridDecrypt.decrypt(
    ciphertext, associatedData);
```

Расшифровка сообщения посредством тех же присоединенных данных. При неудаче выдается исключение

Одно замечание по строке `ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM`: ECIES (Elliptic Curve Integrated Encryption Scheme), или схема шифрования на основе эллиптической кривой, — это стандарт гибридного шифрования, о котором вы узнаете чуть позже. В остальной части строки перечислены алгоритмы инстанцирования данной схемы:

- `P256` — это стандартизованная NIST эллиптическая кривая, представленная в главе 5;
- `HKDF` — функция деривации ключей, рассмотренная в главе 8;
- `HMAC` — имитовставка, представленная в главе 3;
- `SHA256` — хеш-функция, о которой речь шла в главе 2;

- `AES128_GCM` — алгоритм аутентифицированного шифрования AES-GCM с использованием 128-разрядного ключа, представленный в главе 4.

Видите, как все начинает складываться? В следующем разделе мы познакомимся с RSA и ECIES — двумя широко распространенными стандартами асимметричного и гибридного видов шифрования.

6.3. АСИММЕТРИЧНОЕ ШИФРОВАНИЕ ПОСРЕДСТВОМ АЛГОРИТМА RSA: БОЛЬШЕЕ И МЕНЬШЕЕ ЗЛО

Настало время рассмотреть стандарты, определяющие асимметричный и гибридный виды шифрования на практике. С самого начала своего существования оба примитива находились под пристальным вниманием криптоаналитиков, которые обнаруживали в связанных с ними стандартах и реализациях множество уязвимостей и слабых мест. Именно поэтому я начну раздел с представления алгоритма асимметричного шифрования RSA, а затем расскажу о том, как не следует его использовать. В остальной части главы будут рассмотрены реальные стандарты, которые можно применять в ходе работы с асимметричным и гибридным видами шифрования:

- *RSA-OAEP* — основной стандарт для выполнения асимметричного шифрования с использованием алгоритма RSA;
- *ECIES* — основной стандарт для выполнения гибридного шифрования с помощью алгоритма Диффи — Хеллмана на эллиптических кривых.

6.3.1. Классический алгоритм RSA

В этом разделе вы узнаете об алгоритме шифрования с использованием открытого ключа RSA и о многолетней истории его стандартизации. Это полезно для понимания других безопасных схем, основанных на RSA.

К сожалению, с момента своего опубликования в 1977 году алгоритм RSA приобрел довольно плохую репутацию. Одно из популярных мнений заключается в том, что он слишком прост для понимания и разработки, поэтому многие люди изучают его самостоятельно, что приводит к появлению большого количества уязвимых реализаций. Это правильная мысль, но она не полностью передает масштаб проблемы. Мало того что упрощенная реализация концепции RSA (часто называемая *классическим алгоритмом RSA*) небезопасна, так даже сами стандарты были признаны уязвимыми! Однако, чтобы разобраться во всем этом, нужно рассмотреть, как работает RSA.

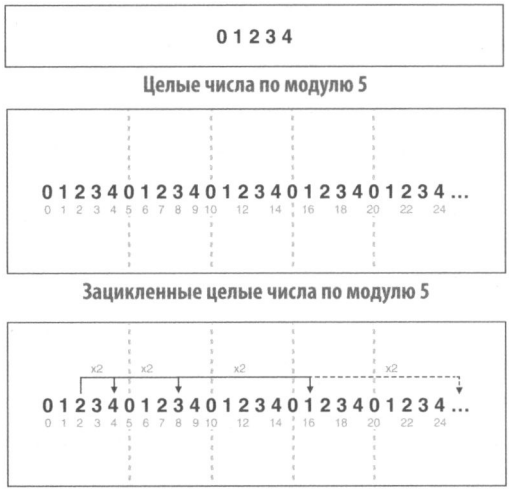
Помните мультипликативную группу чисел по модулю простого числа p? (Мы говорили о ней в главе 5.) Она представляет собой множество строго

положительных целых чисел 1, 2, 3, 4... $p - 1$. Представим, что одно из этих чисел и есть наше сообщение. При достаточно большом значении p, скажем 4096 бит, оно может содержать около 500 символов.

ПРИМЕЧАНИЕ

Сообщение с точки зрения компьютера – это просто набор байтов, которые можно интерпретировать как число.

Мы видели, что, возводя число (допустим, наше сообщение) в степень, можно получить другие числа, образующие *подгруппу* (рис. 6.10).



Если взять в качестве образующего элемента 2, получится подгруппа {2, 4, 3, 1}

Рис. 6.10. Целые числа по модулю простого числа (здесь 5) делятся на различные подгруппы. Выбрав какой-либо элемент в качестве образующего (допустим, число 2) и возведя его в степень, мы можем получить подгруппу. В RSA образующим элементом выступает сообщение

При определении способа шифрования с помощью RSA мы публикуем в открытом доступе экспоненту e (для *шифрования*) и простое число p. (В реальности p не может быть простым числом, но пока не будем обращать на это внимания.) Шифрование сообщения m выполняется с помощью следующего расчета:

$$\text{шифротекст} = m^e \bmod p.$$

Например, расчет для шифрования сообщения $m = 2$ при $e = 2$ и $p = 5$ будет выглядеть так:

$$\text{шифротекст} = 2^2 \bmod 5 = 4.$$

В этом и заключается суть шифрования с помощью RSA!

ПРИМЕЧАНИЕ

Как правило, для быстроты выполнения шифрования в качестве открытой экспоненты *e* выбирается небольшое число. Обычно стандарты и реализации используют простое число 65 537.

Теперь у нас есть возможность шифровать поступающие сообщения. Но как их расшифровать? Помните, что, если продолжать возводить образующий элемент в степень, в конечном счете вернется исходное число (рис. 6.11).

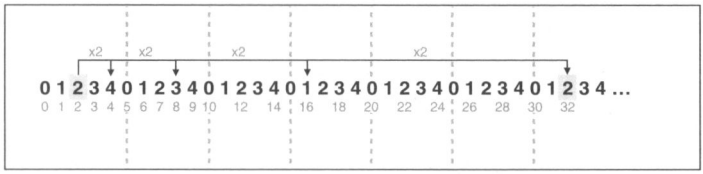


Рис. 6.11. Предположим, что сообщением является число 2. Возведя его в степень, мы можем получить другие числа группы. Сделав это определенное количество раз, вернемся к исходному сообщению — 2. Таким образом, можно сказать, что группа зацикленная. Это свойство может быть использовано для восстановления сообщения после возведения его в степень некоторое количество раз

Данный факт должен дать вам представление о том, как реализовать расшифрование: необходимо выяснить, сколько раз нужно возвести шифротекст в степень, чтобы восстановить исходный образующий элемент, который и является сообщением. Допустим, вам известно такое число, назовем его *секретной экспонентой d* (от слова *decryption* — «расшифрование»). Для того чтобы восстановить сообщение, нужно полученный

$$\text{шифротекст} = \text{сообщение}^e \bmod p$$

возвести в степень d:

$$\text{шифротекст}^d = (\text{сообщение}^e)^d = \text{сообщение}^{ed} = \text{сообщение} \bmod p.$$

Математика нахождения секретной экспоненты d несколько запутана. Если говорить просто, она является результатом вычисления обратного элемента открытой экспоненты *(e)* по модулю порядка (количества элементов) группы:

$$d = e^{-1} \bmod \text{порядок}.$$

Так как у нас есть эффективный алгоритм вычисления обратных элементов по модулю числа, например расширенный алгоритм Евклида, данная задача не вызовет сложностей. Однако есть и другая проблема! Порядок простого числа p равен $p - 1$, что значит: *любой может без труда вычислить секретную экспоненту*, поскольку все элементы этого уравнения, кроме d, находятся в открытом доступе.

ТЕОРЕМА ЭЙЛЕРА

Как мы получили предыдущее уравнение для вычисления секретной экспоненты d? Теорема Эйлера утверждает, что для числа m, взаимно простого по отношению к p, то есть не имеющего с ним общих множителей:

$$m^{порядок} = 1 \bmod p,$$

где порядок — это количество элементов в мультипликативной группе, образованной целыми числами по модулю p. Из этого следует, что для любого целого кратного числа:

$$m^{1 + кратное \times порядок} = m(m^{порядок})^{кратное} \bmod p = m \bmod p.$$

Таким образом, уравнение:

$$m^{ed} = m \bmod p$$

можно преобразовать в:

$$ed = 1 + кратное \times порядок$$

и далее упростить до:

$$ed = 1 \bmod порядок.$$

По определению это означает, что d является элементом, обратным e по модулю *порядка*.

Один из способов предотвратить вычисление секретной экспоненты состоит в том, чтобы скрыть порядок группы. В этом и заключается блестящая идея RSA: если модуль не простое число, а произведение простых чисел $N = pq$, где p и q — большие простые числа, скрытые от доступа, то без этих p и q будет сложно вычислить порядок мультипликативной группы!

ПОРЯДОК ГРУППЫ RSA

Порядок мультипликативной группы по модулю числа N можно вычислить с помощью функции Эйлера $\phi(N)$, которая возвращает количество чисел, *взаимно простых* по отношению к N. Например, 5 и 6 считаются взаимно простыми числами, так как в качестве делителя оба имеют единственное положительное целое число — 1. А вот 10 и 15 не являются таковыми, так как они оба делятся не только на 1, но и на 5. Порядок мультипликативной группы по модулю RSA $N = pq$ равен:

$$\phi(N) = (p - 1)(q - 1).$$

Это уравнение слишком сложно вычислить, если не знать факторизацию N.

Итак, RSA работает следующим образом.

- Для генерации ключа:
  1) генерируем два больших простых числа p и q;
  2) выбираем случайную открытую или фиксированную экспоненту e, например $e = 65\,537$;
  3) открытый ключ — это открытая экспонента e и открытый модуль $N = pq$;
  4) выводим секретную экспоненту $d = e^{-1} \bmod (p - 1)(q - 1)$;
  5) закрытым ключом является секретная экспонента d.
- Для шифрования вычисляем *сообщение*$^e \bmod N$.
- Для расшифрования вычисляем *шифротекст*$^d \bmod N$.

Схему работы алгоритма RSA на практике можно увидеть на рис. 6.12.



Рис. 6.12. RSA-шифрование осуществляется возведением числа (сообщения) в открытую степень e по открытому модулю N = pq. RSA-расшифрование выполняется возведением зашифрованного числа в секретную степень d по открытому модулю N

В основе алгоритма RSA лежит *задача факторизации*. Не зная значений p и q, никто не сможет вычислить порядок группы. Таким образом, посторонний не сможет вывести секретную экспоненту из открытой экспоненты, в этом RSA схож с алгоритмом Диффи — Хеллмана и его задачей дискретного логарифма (рис. 6.13).

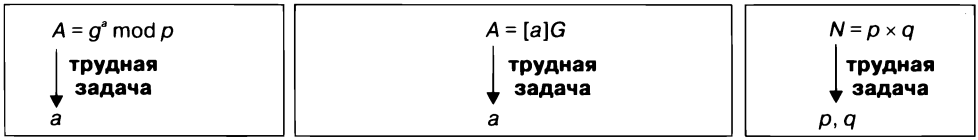


Рис. 6.13. Алгоритмы Диффи — Хеллмана, Диффи — Хеллмана на эллиптических кривых и RSA — это асимметричные алгоритмы, в основе которых лежат три различные математические задачи, считающиеся трудноразрешимыми. Трудными являются те задачи, для которых нет известных эффективных алгоритмов решения при оперировании большими числами

Таким образом, классический алгоритм RSA работает по модулю составного числа $N = pq$, где p и q — два больших простых числа, которые должны оставаться

в секрете. Теперь, когда вы поняли, как работает RSA, давайте посмотрим, насколько низка его криптостойкость на практике и как стандарты исправили данную проблему.

6.3.2. Почему не стоит использовать RSA PKCS#1 v1.5

По многим причинам классический алгоритм RSA в своем оригинальном виде имеет низкую криптостойкость. Прежде чем приступить к рассмотрению безопасной версии RSA, поговорим о том, чего следует избегать.

Существует множество причин, почему нельзя использовать классический RSA в его первозданном виде. Одна из них заключается в том, что если вы шифруете небольшие сообщения, допустим $m = 2$, то злоумышленник может зашифровать все небольшие числа, например от 0 до 100, и быстро проследить, совпадает ли какое-то из них с вашим шифротекстом. Если совпадает, то он узнает, что вы зашифровали.

Эта проблема решается стандартами, делающими сообщения слишком большими для такого перебора. В частности, они увеличивают размер сообщения (до шифрования) с помощью *недетерминированного* дополнения. Например, на рис. 6.14 показан стандарт RSA PKCS#1 v1.5, который дополняет сообщение некоторым количеством случайных байтов.

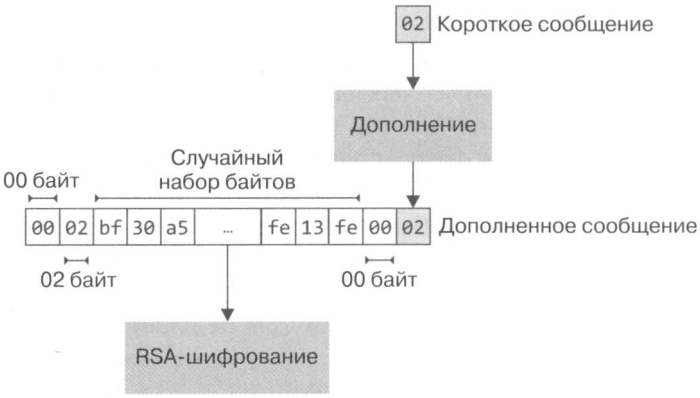


Рис. 6.14. Стандарт RSA PKCS#1 v1.5 устанавливает дополнение к сообщению перед шифрованием. Оно должно носить обратимый характер, чтобы при расшифровке можно было от него избавиться, и добавлять к сообщению достаточное количество случайных байтов, чтобы защитить его от атак, проводимых методом грубой силы

Стандарт PKCS#1 – это фактически первый стандарт на базе алгоритма RSA, опубликованный в рамках серии документов Public Key Cryptography Standard (PKCS), написанных компанией RSA в начале 1990-х годов. Несмотря на то что в нем были исправлены некоторые известные проблемы алгоритма, в 1998 году Блейхенбахер обнаружил практическую атаку на PKCS#1 v1.5, позволявшую расшифровывать сообщения, которые были дополнены согласно этому стандарту.

Эта атака стала печально известной под названием «атака миллионами сообщений», поскольку ее проведение требовало миллиона сообщений. Хотя позднее появились методы защиты от данной уязвимости, она никуда не исчезла, так как, по мнению исследователей, безопасная реализация этих методов — слишком сложная задача, если вообще возможная.

АТАКА НА ОСНОВЕ АДАПТИВНО ПОДОБРАННОГО ШИФРОТЕКСТА

Атака миллионом сообщений Блейхенбахера представляет собой разновидность атаки, называемой в теоретической криптографии *атакой на основе адаптивно подобранного шифротекста* (adaptive chosen-ciphertext attacks, CCA2). При ее реализации злоумышленник подает различные сообщения, зашифрованные алгоритмом RSA *(подобранные шифротексты)*, и смотрит, как они расшифровываются. Он продолжает атаку, *адаптируясь* на основании предыдущих наблюдений. CCA2 часто используется для моделирования атак в доказательствах криптографической устойчивости.

Возможность проведения данной атаки обусловлена *податливостью* шифротекстов RSA. Дело в том, что можно подделать RSA-шифротекст так, что это не повлияет на его расшифровку негативно. Если взломщик видит шифротекст $c = m^e \bmod N$, он может подать вот такой вариант:

$$3^e \times m^e = (3m)^e \bmod N,$$

который после расшифровки будет иметь вид:

$$((3m)^e)^d = (3m)^{e \times d} = 3m \bmod N.$$

В качестве примера я взял число 3, но исходное сообщение можно умножить на любое число. На практике сообщение должно быть правильно сформировано с помощью дополнения, чтобы в случае его подделки расшифровка не удалась. Тем не менее иногда случается так, что даже при злонамеренном изменении шифротекста дополнение признается действительным после расшифровки.

Это свойство и было использовано Блейхенбахером в атаке миллионом сообщений на алгоритм RSA PKCS#1 v1.5. Атака заключается в перехвате и изменении зашифрованного сообщения и отправке его тому, кто отвечает за его расшифровку. Наблюдая за тем, сможет ли этот человек расшифровать его (дополнение осталось действительным), атакующий получает некоторую информацию о диапазоне сообщения. Поскольку первые два байта — это 0x0002, он знает, что расшифровка меньше некоторого значения. Выполняя эти действия итеративно, можно сузить данный диапазон до исходного сообщения.

Несмотря на то что атака Блейхенбахера широко известна, алгоритм RSA PKCS#1 v1.5 до сих пор используется во множестве систем шифрования сообщений. Работая консультантом по безопасности, я обнаружил множество приложений, уязвимых к этой атаке, так что будьте осторожны!

6.3.3. Асимметричное шифрование с помощью RSA-OAEP

В 1998 году была выпущена версия 2.0 стандарта PKCS#1. Она содержала новую схему дополнения для RSA под названием *«оптимальное асимметричное шифрование с дополнением»* (Optimal Asymmetric Encryption Padding, OAEP). В отличие от своего предшественника PKCS#1 v1.5, OAEP неуязвима для атаки Блейхенбахера и поэтому в настоящее время является сильным стандартом RSA-шифрования. Рассмотрим, как работает OAEP и как он предотвращает атаку миллионом сообщений.

Прежде всего отмечу, что, как и большинство криптографических алгоритмов, OAEP идет с алгоритмом генерации ключа. Для работы ему нужен параметр безопасности (рис. 6.15).

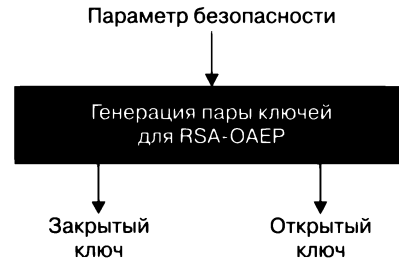


Рис. 6.15. Работая с RSA-OAEP, как и со многими другими алгоритмами с открытым ключом, сначала необходимо сгенерировать пару ключей, которая в дальнейшем может быть использована в других алгоритмах, предоставляемых криптографическим примитивом

Данный алгоритм принимает некоторое количество битов в качестве параметра безопасности. Как и в случае с алгоритмом Диффи – Хеллмана, операции выполняются в ряде чисел по модулю большого числа. Говоря о безопасности реализации RSA, мы обычно имеем в виду размер этого модуля, что, если вы помните, походит на принцип работы алгоритма Диффи – Хеллмана.

В настоящее время, по расчетам большинства руководств (см. https://keylength.com), 128-битная защита обеспечивается модулем от 2048 до 4096 бит. Из-за существенных различий методов этих расчетов большинство приложений, по-видимому, решили остановиться на параметрах размером 4096 бит.

ПРИМЕЧАНИЕ

Модуль большого числа в алгоритме RSA – это не простое число, а произведение двух больших простых чисел p и q ($N = pq$). Для получения модуля в 4096 бит алгоритм генерации ключей в большинстве случаев делит это число пополам и генерирует оба числа, p и q, примерно 2048 бит.

При шифровании алгоритм сначала дополняет сообщение битами и смешивает его со случайно сгенерированным числом. Затем полученный результат шифруется с помощью алгоритма RSA. При расшифровке шифротекста данный процесс выполняется в обратном порядке (рис. 6.16).

RSA-OAEP использует смешивание, чтобы гарантировать, что злоумышленник не получит никакой информации в случае утечки нескольких зашифрованных с помощью алгоритма RSA битов. Удаление OAEP-дополнения возможно только в том случае, если получены все или почти все байты дополненного открытого текста! Кроме того, данный алгоритм устойчив к атаке Блейхенбахера, поскольку

делает невозможным получение правильно сформированного открытого текста путем внесения изменений в шифротекст.

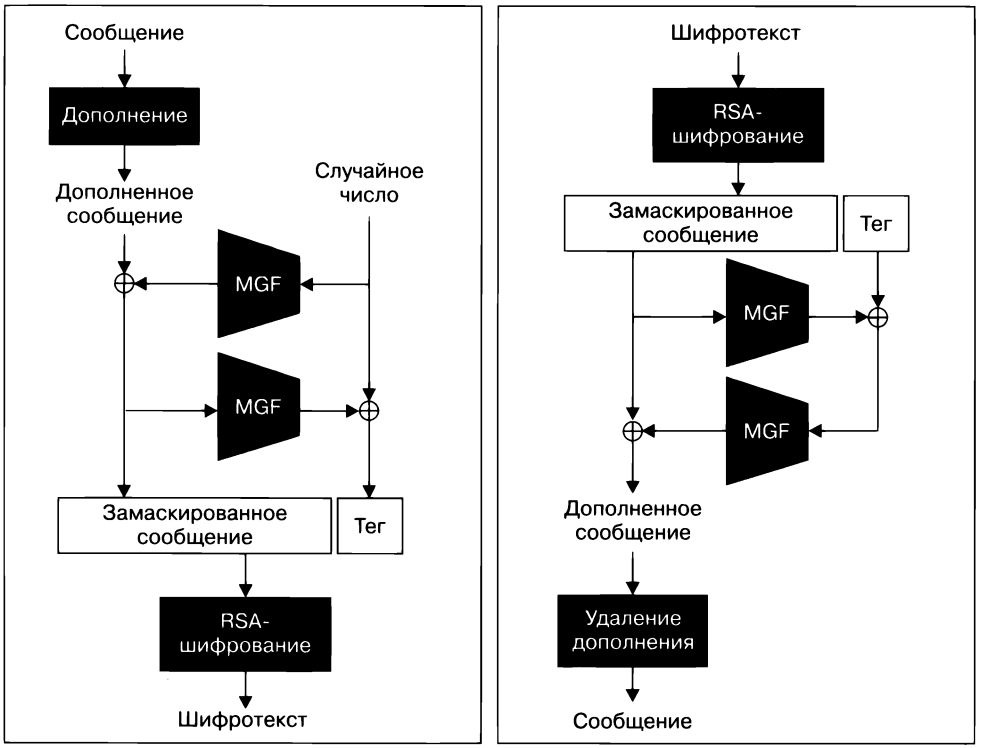


Рис. 6.16. RSA-OAEP работает посредством смешивания сообщения со случайным числом перед шифрованием и возвращения его в исходный вид после расшифровки. В центре алгоритма находится функция генерации масок (MGF), которая используется для получения случайного вида и увеличения либо уменьшения размера входных данных

ПРИМЕЧАНИЕ

Осведомленность об открытом тексте — это свойство, которое означает, что злоумышленнику слишком сложно создать такой шифротекст, который был бы успешно расшифрован (разумеется, без помощи шифрования). Благодаря этому свойству схема OAEP устойчива к атаке Блейхенбахера.

Аббревиатура MGF в контексте OAEP означает *функцию генерации маски*. На практике MGF — это функция расширенного выхода (XOF), о которой мы говорили в главе 2. Поскольку MGF появилась раньше, чем XOF, то для ее построения используется хеш-функция, которая многократно хеширует входные данные с помощью счетчика (рис. 6.17). Вот так и работает OAEP!

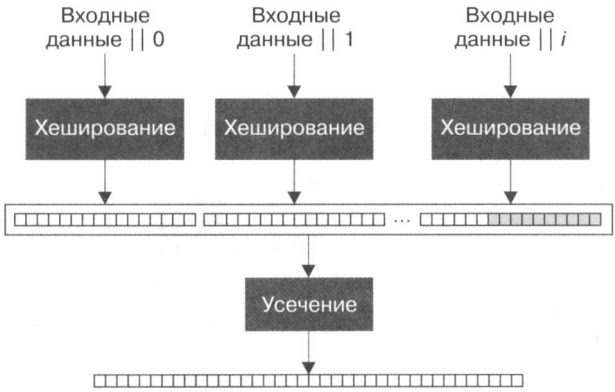


Рис. 6.17. Функция генерации масок (MGF) принимает на вход файл произвольной длины и выдает на выходе псевдослучайный файл произвольной длины. Она работает путем хеширования входных данных и счетчика, конкатенации дайджестов и усечения результата для получения нужной длины

АТАКА ОРАКУЛА С ДОПОЛНЕНИЕМ

Спустя всего три года после выхода стандарта OAEP Джеймс Мангер обнаружил, что неправильная реализация делает данную схему уязвимой к временной атаке, схожей с атакой миллионом сообщений Блейхенбахера, возможность проведения которой гораздо реальнее. К счастью, по сравнению с PKCS#1 v1.5 OAEP гораздо проще реализовать безопасно, что значительно понижает вероятность возникновения уязвимостей в этой схеме.

При этом стоит отметить, что схема OAEP отнюдь не совершенна. За прошедшие годы были предложены и стандартизированы конструкции лучшего качества. Например, RSA-KEM имеет более убедительные доказательства криптостойкости, к тому же она гораздо проще в реализации. На рис. 6.18 можно увидеть, насколько более элегантна данная конструкция.

Обратите внимание на используемую здесь функцию получения ключа (key derivation function, KDF). Это еще один криптографический примитив, который может быть заменен на MGF или XOF. Подробнее о том, что такое KDF, я расскажу в главе 8, посвященной криптографической случайности и секретам.

В настоящее время, когда в протоколах и приложениях задействуется RSA, в большинстве случаев это либо небезопасный PKCS#1 v1.5, либо OAEP. Впрочем, как при обмене ключами, так и при гибридном шифровании все больше протоколов отказываются от RSA-шифрования в пользу алгоритма Диффи — Хеллмана на эллиптических кривых (ECDH). Это вполне объяснимо, поскольку ECDH предоставляет более короткие открытые ключи и в целом имеет гораздо лучшие стандарты и более безопасные реализации.

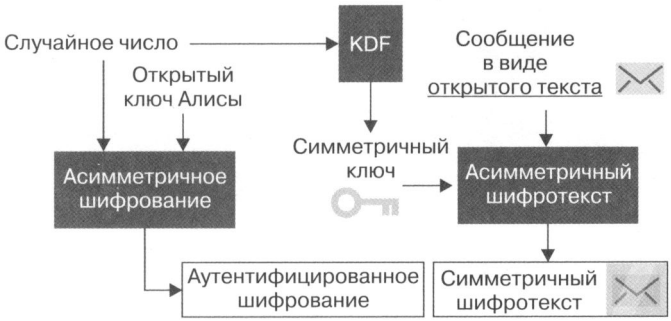


Рис. 6.18. RSA-KEM — это схема шифрования случайного числа, полученного с помощью RSA, не требующая какого-либо дополнения. Мы можем пропустить это случайное число через функцию получения ключа (KDF) и получить симметричный ключ, который в дальнейшем будет использован в алгоритме аутентифицированного шифрования

6.4. ГИБРИДНОЕ ШИФРОВАНИЕ С ПОМОЩЬЮ АЛГОРИТМА ECIES

Хотя существует множество схем гибридного шифрования, наиболее широко распространенный стандарт — это *схема гибридного шифрования на основе эллиптической кривой* (Elliptic Curve Integrated Encryption Scheme, ECIES). Она была разработана для применения в связке с ECDH и включена во многие стандарты: ANSI X9.63, ISO/IEC 18033-2, IEEE 1363a и SECG SEC 1. К сожалению, различные криптографические библиотеки по-разному реализуют гибридное шифрование. Это отчасти объясняется тем, что каждый стандарт реализует свой вариант.

По этой причине две одинаковые реализации гибридного шифрования встречаются редко. Важно понимать: несмотря на некоторое неудобство, если все участники протокола используют один и тот же вариант или документируют детали примененной схемы гибридного шифрования, то проблем не будет.

ECIES работает аналогично гибридной схеме шифрования, описанной в разделе 6.2. Отличие заключается в том, что мы реализуем часть KEM не с помощью примитива асимметричного шифрования, а посредством обмена ключами ECDH. Объясню пошагово.

Во-первых, чтобы зашифровать сообщение Алисе, используется обмен ключами на основе алгоритма (EC)DH с открытым ключом Алисы и парой ключей, сгенерированной специально для этого случая, — *эфемерной парой ключей*. Затем полученный общий секрет применяется в алгоритме аутентифицированного шифрования, например AES-GCM, для шифрования более длинного сообщения (рис. 6.19).

После этого можно отправить Алисе эфемерный открытый ключ и шифротекст. Далее она может использовать этот ключ для обмена ключами со своей собственной парой ключей, а результат обмена — для расшифровки шифротекста и получения исходного сообщения. В итоге получится либо исходное сообщение, либо ошибка, если открытый ключ или зашифрованное сообщение подделаны при передаче (рис. 6.20).

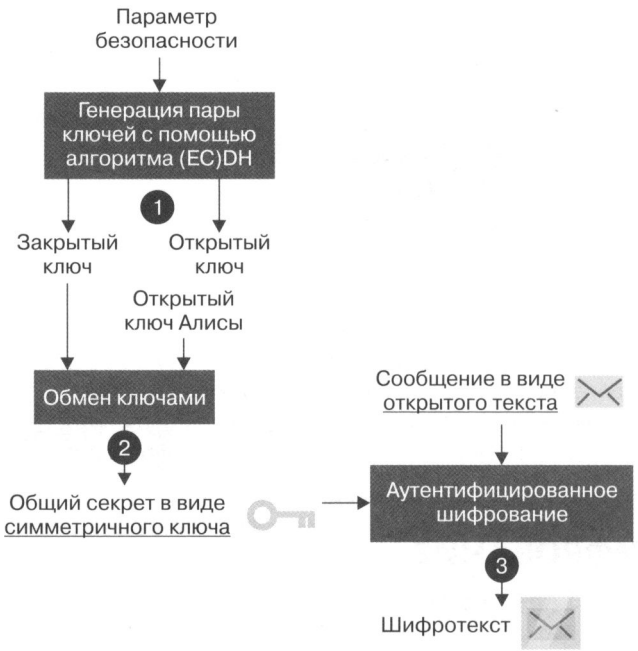


Рис. 6.19. Чтобы зашифровать сообщение для Алисы с помощью гибридного шифрования на основе (EC)DH, генерируется эфемерная пара ключей Диффи — Хеллмана на эллиптической кривой *(1)*. Затем выполняется обмен ключами с эфемерным закрытым ключом отправителя и открытым ключом Алисы *(2)*. Полученный общий секрет выступает в качестве симметричного ключа для алгоритма аутентифицированного шифрования, посредством которого зашифровывается сообщение *(3)*

В целом именно так работает ECIES. Существует также вариант ECIES с использованием алгоритма Диффи — Хеллмана, названный IES, который работает практически так же, но не особо популярен.

УСТРАНЕНИЕ ОШИБКИ ВЫБОРКИ В РЕЗУЛЬТАТЕ ОБМЕНА КЛЮЧАМИ

Стоит отметить, что процесс, изображенный на рис. 6.20, был упрощен. Большинство примитивов аутентифицированного шифрования предполагают наличие неравномерно распределенного случайного симметричного ключа. Поскольку результат обмена ключами, как правило, не является таковым, нам необходимо предварительно пропустить общий секрет через KDF или XOF (см. главу 2). Подробнее об этом будет рассказано в главе 8.

Неравномерно распределенная случайная величина здесь означает, что статистически некоторые биты результата обмена ключами чаще могут быть равны 0, нежели 1, или наоборот. Например, первые биты могут быть всегда 0.

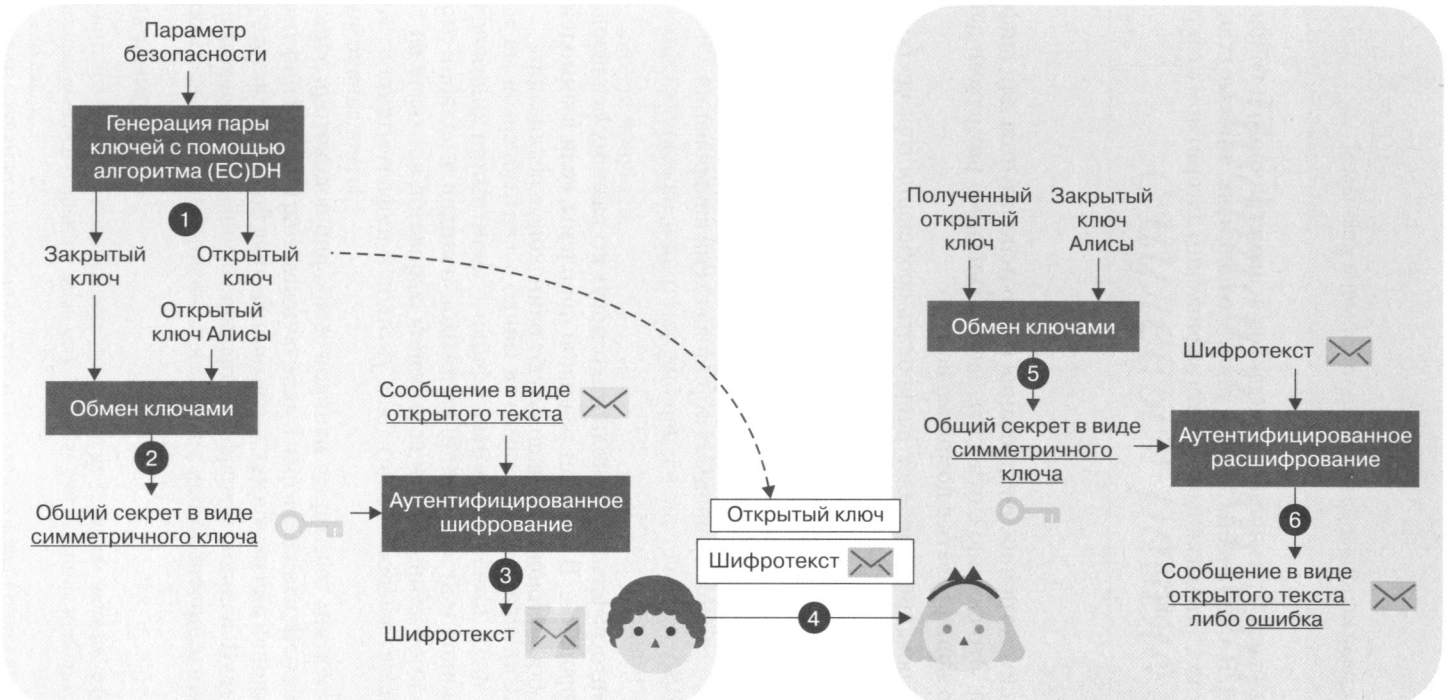


Рис. 6.20. Продолжение рис. 6.19: после того как Алиса получила эфемерный открытый ключ отправителя и зашифрованное сообщение *(4)*, она может произвести обмен ключами *(5)*, используя свой закрытый ключ и эфемерный открытый ключ. Полученный общий секрет применяется в качестве симметричного ключа для расшифровки сообщения посредством того же аутентифицированного алгоритма шифрования *(6)*

УПРАЖНЕНИЕ

Почему нельзя сразу использовать результат обмена ключами?

Вот и все, что касается различных стандартов асимметричного и гибридного видов шифрования. В следующей главе вы узнаете о подписях, которые станут последним и, возможно, самым важным криптографическим алгоритмом с открытым ключом, изложенным в части I.

РЕЗЮМЕ

- Само по себе асимметричное шифрование редко используется для шифрования сообщений. Это связано с относительно небольшими размерами данных, которые можно зашифровать с помощью него.
- Гибридное шифрование позволяет шифровать сообщения гораздо большего размера за счет сочетания асимметричного шифрования (или обмена ключами) с симметричным алгоритмом аутентифицированного шифрования.
- Стандарт RSA PKCS#1 v1.5 в большинстве случаев не подходит для асимметричного шифрования. Предпочтительнее применять алгоритм RSA-OAEP, стандартизированный в RSA PKCS#1 v2.2.
- ECIES — широко распространенная гибридная схема шифрования. Она предпочтительнее RSA-схем за счет размера параметров и надежных стандартов, лежащих в ее основе.
- Различные криптографические библиотеки могут по-разному реализовывать гибридное шифрование. В этом нет реальной проблемы, если взаимодействующие между собой приложения используют одинаковые реализации.

7 Подписи и доказательства с нулевым разглашением

В этой главе

- ✓ Доказательства с нулевым разглашением и криптографические подписи.
- ✓ Существующие стандарты криптографической подписи.
- ✓ Неочевидные особенности подписей и способы избежать связанных с ними ловушек.

Сейчас вы познакомитесь с одним из самых распространенных и мощных криптографических примитивов — с цифровыми подписями. Если говорить простым языком, цифровые подписи похожи на привычные нам подписи, которые люди ставят на чеках и в договорах, конечно, за исключением того, что они относятся к области криптографии и поэтому дают гораздо больше гарантий, чем их рукописные эквиваленты.

В мире протоколов цифровые подписи открывают множество возможностей, поэтому они будут играть важную роль в части II книги. В этой главе я расскажу, что представляет собой данный примитив, как его можно использовать в реальном мире и каковы современные стандарты цифровой подписи. Будут затронуты также вопросы безопасности и риски, связанные с использованием цифровых подписей.

ПРИМЕЧАНИЕ

Криптографические подписи часто называют *цифровыми подписями* (digital signatures) или *схемами подписи* (signature schemes)[1]. В книге эти термины используются как синонимы.

[1] Нередки случаи использования термина «электронная подпись». — *Примеч. пер.*

Для полноты понимания этой главы рекомендую прочитать:

- главу 2 о хеш-функциях;
- главу 5 об обмене ключами;
- главу 6 об асимметричном шифровании.

7.1. ЧТО ТАКОЕ ПОДПИСЬ

В главе 1 я говорил, что криптографические подписи очень похожи на подписи из реальной жизни. По этой причине они обычно являются одним из самых интуитивно понятных криптографических примитивов.

- Только вы можете поставить свою подпись под каким-либо сообщением.
- Любой человек может проверить вашу подпись, добавленную к сообщению.

Поскольку мы находимся в области асимметричной криптографии, вы, вероятно, догадываетесь, как выглядит эта асимметрия в контексте подписей. Схема подписи обычно состоит из трех алгоритмов:

- алгоритма генерации новой пары ключей — закрытого и открытого (открытый ключ может быть передан любому лицу);
- алгоритма подписания, который для создания подписи использует закрытый ключ и сообщение;
- алгоритма проверки, который обрабатывает открытый ключ, сообщение и подпись и возвращает оповещение об успехе или ошибке.

Иногда закрытый ключ называют также *ключом подписи* (signing key), а открытый ключ — *ключом проверки* (verifying key). Логично, правда? Вот как выглядят в общих чертах эти три алгоритма (рис. 7.1).

Для чего нужны подписи? Они хорошо подходят для проверки подлинности источника и целостности сообщения.

- *Источник* (origin) — если на сообщении стоит моя подпись, значит, оно пришло от меня.
- *Целостность* (integrity) — если кто-то изменит сообщение, подпись станет недействительной.

ПРИМЕЧАНИЕ

Хотя эти два свойства связаны с аутентификацией, их часто разделяют на *аутентификацию источника* и *аутентификацию сообщения* (или целостность).

В некотором смысле подписи похожи на имитовставки (MAC) из главы 3. Но в отличие от MAC они позволяют проверять подлинность сообщений асим-

метрично: получатель может убедиться в том, что сообщение не было подделано, без закрытого ключа (ключа подписи). Далее я покажу, как эти алгоритмы можно использовать на практике.

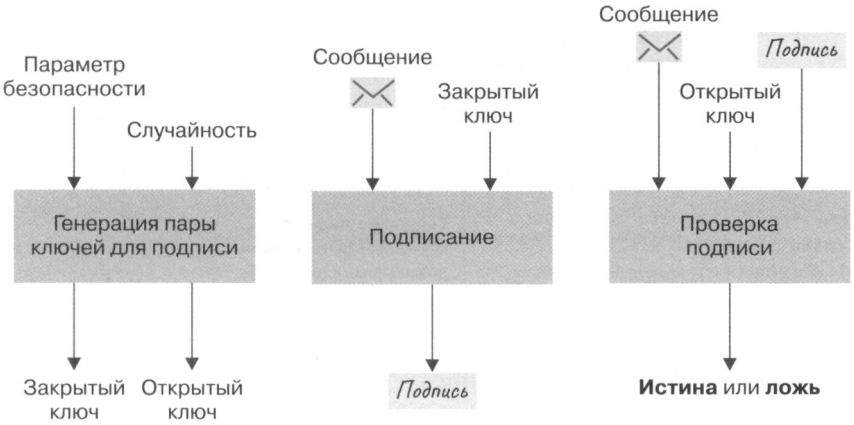


Рис. 7.1. Внутренняя система цифровой подписи. Как и в других криптографических алгоритмах с открытым ключом, сначала необходимо сгенерировать пару ключей на основе параметра безопасности и некоторой случайности. Затем можно использовать алгоритм подписания с закрытым ключом для подписания сообщения и алгоритм проверки с открытым ключом для подтверждения этой подписи. Невозможно подделать подпись, подтверждающую открытый ключ без доступа к соответствующему закрытому ключу

УПРАЖНЕНИЕ

Как вы видели в главе 3, аутентификационные теги, создаваемые MAC, должны проверяться за одно и то же время во избежание атак по времени. Как вы думаете, нужно ли делать то же самое для проверки подписей?

7.1.1. Как на практике происходит добавление и проверка подписи

Рассмотрим практический пример добавления и проверки подписи на хорошо зарекомендовавшей себя библиотеке Python pyca/cryptography (https://cryptography.io). В листинге 7.1 показано, как происходит генерация пары ключей, подписание сообщения с помощью закрытого ключа и применение открытого ключа для проверки подписи.

Листинг 7.1. Добавление и проверка подписи в Python

```
from cryptography.hazmat.primitives.asymmetric.ed25519 import (
    Ed25519PrivateKey
)

private_key = Ed25519PrivateKey.generate()
public_key = private_key.public_key()

message = b"example.com has the public key 0xab70..."
signature = private_key.sign(message)

try:
    public_key.verify(signature, message)
    print("valid signature")
except InvalidSignature:
    print("invalid signature")
```

Используется популярная схема подписи: алгоритм Ed25519

Сначала генерируется закрытый ключ, а затем — открытый

С помощью закрытого ключа подписывается сообщение

С помощью открытого ключа проверяется подпись, добавленная к сообщению

Как я уже говорил, цифровые подписи имеют множество вариантов применения в реальном мире. Рассмотрим один из примеров.

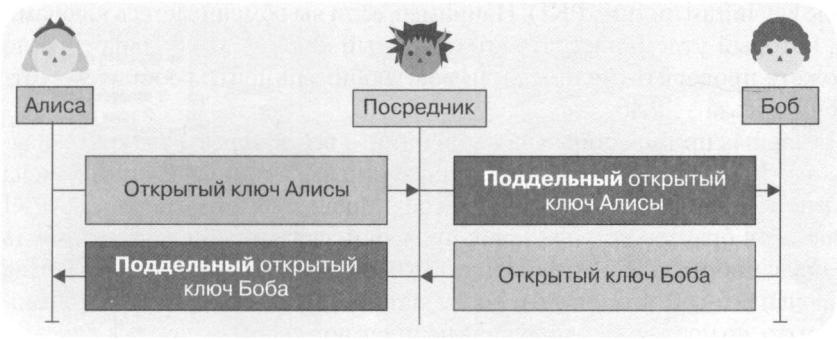
7.1.2. Основной пример применения подписей — аутентифицированный обмен ключами

В главах 5 и 6 были представлены различные способы обмена ключами. Там же вы узнали, что обмен ключами полезен для согласования общего секрета, который затем может быть использован в каком-нибудь аутентифицированном алгоритме шифрования для защиты связи. Однако данный примитив не полностью решает проблему установки безопасного соединения между двумя участниками, поскольку активный посредник может запросто выдать себя за обе стороны обмена. Именно здесь на помощь приходят подписи.

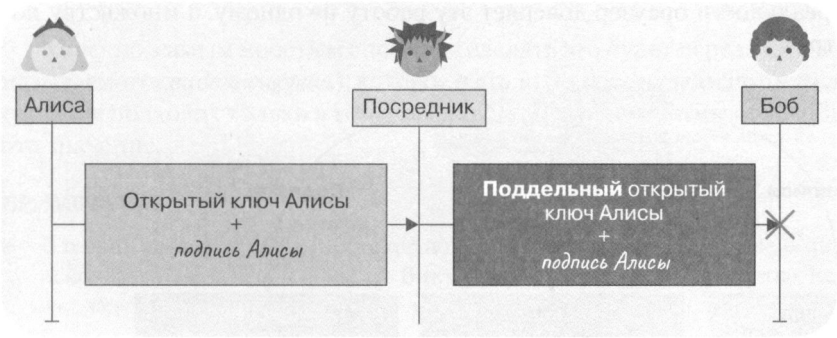
Представьте, что Алиса и Боб пытаются установить между собой защищенный канал связи и что Бобу известен ключ проверки, принадлежащий Алисе. Зная это, Алиса может аутентифицировать свою сторону обмена ключами: она генерирует пару ключей, подписывает открытый ключ своим ключом подписи и отправляет его Бобу. Боб может проверить подлинность подписи с помощью известного ему ключа проверки, а затем использовать открытый ключ для обмена ключами.

Такой обмен ключами называется *аутентифицированным*. Если подпись не проходит проверку, Боб может заключить, что кто-то активно проводит атаку посредника. Аутентифицированный обмен ключами наглядно продемонстрирован на рис. 7.2.

Обратите внимание на то, что в этом примере обмен ключами аутентифицирован только с одной стороны: Алису нельзя сымитировать, а Боба — можно. Если обе стороны аутентифицированы (то есть когда и Боб подписывает свой открытый ключ), мы называем такой обмен *взаимно аутентифицированным*. Понятно, что сейчас полезность подписей при обмене ключами довольно сомнительна, ведь мы заменили проблему незнания открытого ключа Алисы проблемой незнания ее ключа проверки. В следующем разделе будет показано реальное применение аутентифицированного обмена ключами, которое имеет гораздо больше смысла.



Неаутентифицированный обмен ключами



Аутентифицированный обмен ключами

Рис. 7.2. На верхнем фрагменте представлен неаутентифицированный обмен ключами. Он уязвим перед активным посредником, который может без проблем выдать себя за обе стороны обмена, подменив их открытые ключи своими собственными. На нижнем изображен начальный этап обмена ключами, который был аутентифицирован Алисой путем добавления подписи к своему открытому ключу. После подделки сообщения посредником Боб, который знает ключ проверки, принадлежащий Алисе, не может удостоверить подлинность подписи и поэтому прерывает обмен ключами

7.1.3. Использование подписей в реальном мире. Инфраструктура открытых ключей

Подписи становятся гораздо мощнее, если предположить, что доверие является транзитивным. Под этим я подразумеваю, что если вы доверяете мне, а я доверяю Алисе, то вы можете доверять Алисе. Она классная.

Транзитивность доверия позволяет очень сильно менять масштаб доверия в системах. Представьте, что вы доверяете некой авторитетной организации (удостоверяющему центру) и ее ключу проверки. Этот центр подписал сообщения, указывающие открытый ключ Чарльза, открытый ключ Дэвида и т. д. Таким образом, высока вероятность того, что вы будете доверять составленному списку! Такая схема работы с открытым ключом называется *инфраструктурой открытых*

ключей (public key infrastructure, PKI). Например, если вы обмениваетесь ключами с Чарльзом, который утверждает, что его открытый ключ — это большое число 3848... то можете проверить, подписал ли ваш любимый центр сообщение «открытый ключ Чарльза — 3848...».

Одно из реальных применений этой концепции — веб-инфраструктура открытых ключей (веб-PKI). Именно ее использует браузер для аутентификации обмена ключами со множеством популярных веб-сайтов. Упрощенное объяснение веб-PKI (рис. 7.3) таково: в браузер встроен ключ проверки, связанный с каким-нибудь центром. В обязанности последнего входит подписание огромного количества открытых ключей веб-сайтов, которым пользователь сможет доверять. От пользователя это скрыто, но прежде, чем получить подпись под своим открытым ключом, веб-сайты доказывают центру, что они действительно владеют своим доменным именем. (В реальности браузер доверяет эту работу не одному, а множеству подобных центров.)



Рис. 7.3. В веб-PKI браузеры доверяют центру удостоверять соответствие доменов с открытыми ключами. При безопасном посещении веб-сайта браузер может проверить его открытый ключ (то, что он не создан каким-нибудь посредником) с помощью подписи, полученной от центра

Получив общее представление о подписях, давайте углубимся в особенности их работы. Однако сначала нужно сделать отступление и взглянуть на так называемые доказательства с нулевым разглашением (zero-knowledge proofs, ZKP).

7.2. ДОКАЗАТЕЛЬСТВА С НУЛЕВЫМ РАЗГЛАШЕНИЕМ. ПРОИСХОЖДЕНИЕ ПОДПИСЕЙ

Лучший способ разобраться в том, как работают подписи в криптографии, — это понять, откуда они взялись. По этой причине давайте вкратце познакомимся с ZKP, а затем вернемся к подписям.

Представьте: Пегги хочет доказать что-то Виктору[1]. Например, что знает дискретный логарифм по основанию некоторого элемента группы, или, другими словами, значение x в уравнении $Y = g^x$, где g — образующий элемент некоторой группы.

[1] Пегги и Виктор — от Prover и Verifier соответственно. — *Примеч. пер.*



Конечно, самым простым способом сделать это будет передача значения x (так называемого *свидетельства*), которое и стало бы *доказательством знания*. Данное решение подходит только в том случае, если Пегги не против, чтобы Виктор узнал это значение.

ПРИМЕЧАНИЕ

В теории мы говорим, что протокол доказательства является *полным*, если Пегги может использовать его, чтобы доказать Виктору, что она знает свидетельство. Если она не может задействовать его для этой цели, то схема бесполезна, верно?

В криптографии нас больше всего интересуют доказательства знания, которые не раскрывают свидетельство проверяющему. Такие доказательства называются *доказательствами с нулевым разглашением* (zero-knowledge proofs, ZKP).

7.2.1. Протокол идентификации Шнорра — интерактивное доказательство с нулевым разглашением

На следующих страницах мы постепенно выстроим понимание ZKP на основе уязвимых протоколов. Таким образом можно наглядно показать, как Пегги доказывает, что она знает x, не раскрывая его.

Типичный криптографический способ решения такой задачи — спрятать значение с помощью случайности, например, зашифровав его. Однако мы не просто прячем значение — нам нужно доказать, что оно существует. Для этого понадобится алгебраический способ маскировки. Тривиальным решением является добавление к свидетельству случайно сгенерированной величины k:

$$s = k + x.$$

Так, Пегги может отправить Виктору замаскированное свидетельство s с добавленной к нему случайной величиной k. Пока что у Виктора нет оснований верить в то, что Пегги действительно спрятала свидетельство в s. Ведь если она не знает свидетельство x, тогда s, вероятно, будет просто случайным значением. Виктор знает, что x прячется в экспоненте g, так как он знает $Y = g^x$.

Для того чтобы убедиться в том, что Пегги действительно знает свидетельство, Виктор может проверить, совпадает ли то, что она ему передала, с тем, что знает он. Проверку нужно выполнить в части с экспонентой g, ведь именно там находится свидетельство. Другими словами, Виктору необходимо проверить, равны ли следующие два значения:

- $g^s (= g^{k+s})$;
- $Yg^k (= g^x g^k = g^{x+k})$.

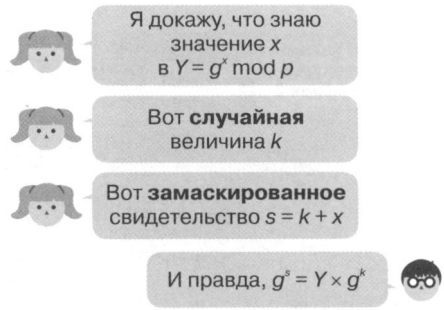


Рис. 7.4. Для того чтобы доказать Виктору, что она знает свидетельство x, Пегги маскирует его (добавляя к нему случайную величину k) и вместо x отправляет замаскированное свидетельство s

Суть заключается в том, что только тот, кто знает свидетельство x, мог сформировать замаскированное свидетельство s, удовлетворяющее этому уравнению. И как таковое это является доказательством знания. Система ZKP кратко изложена на рис. 7.4.

У этой схемы есть одна проблема: она явно небезопасна! Поскольку уравнение, маскирующее x, имеет только одно неизвестное (само x), Виктору не составит труда вывести его:

$$x = s - k.$$

Чтобы избежать данной проблемы, Пегги может спрятать саму случайную величину k! На этот раз придется скрыть ее в экспоненте (вместо того чтобы добавить к другой случайной величине) и убедиться, что уравнение Виктора по-прежнему работает:

$$R = g^k.$$

Таким образом, Виктор не узнает значение k (это задача дискретного логарифма из главы 5) и, следовательно, не сможет восстановить свидетельство x. Тем не менее у него есть достаточно информации, чтобы удостовериться: Пегги знает x! Виктору остается проверить, что $g^s (= g^{k+x} = g^k g^x)$ равно $YR (= g^x g^k)$. На рис. 7.5 изображена вторая попытка применения протокола ZKP.

Есть еще одна проблема в нашей схеме: Пегги может обмануть. Она может убедить Виктора в том, что знает x, не зная x! Все, что ей нужно сделать, — обратить шаг, на котором она вычисляет свое доказательство. Сначала она генерирует случайную величину s, а затем на ее основе вычисляет значение R:

$$R = g^s Y^{-1}.$$

Виктор, в свою очередь, вычисляет $YR = Yg^s Y^{-1}$, что действительно соответствует g^s. (Трюк Пегги с использованием обратной величины для вычисления значения применяется во многих атаках в криптографии.)

ПРИМЕЧАНИЕ

В теории мы говорим, что схема корректна, если у Пегги нет возможности для обмана (если она не знает x, то не может обмануть Виктора).

При корректном протоколе ZKP Виктор должен гарантировать, что Пегги вычислит s из R, а не наоборот. Для этого он делает протокол *интерактивным*.

1. Пегги должна зафиксировать свою случайную величину k как обязательство и таким образом лишить себя возможности менять ее в дальнейшем.
2. Получив обязательство Пегги, Виктор вносит в протокол собственную случайность. Он генерирует случайную величину c *(вызов)* и отправляет ее Пегги.
3. Затем на основе случайной величины k и вызова c Пегги может вычислить свое скрытое обязательство.

ПРИМЕЧАНИЕ

В главе 2 рассказывалось о схемах обязательств: в ней мы использовали хеш-функцию для фиксации значения как обязательства, которое позже можем раскрыть. Однако схемы обязательств, основанные на хеш-функциях, не позволяют выполнять интересные арифметические действия над скрытым значением. Вместо этого мы можем просто довести образующий элемент до значения g^k, что уже и делаем.

Поскольку Пегги не может выполнить последний шаг без вызова c, а Виктор не отправит его ей, не получив зафиксированной случайной величины k, Пегги вынуждена вычислять s на основе k. Полученный протокол (рис. 7.6) часто называют *протоколом идентификации Шнорра*.

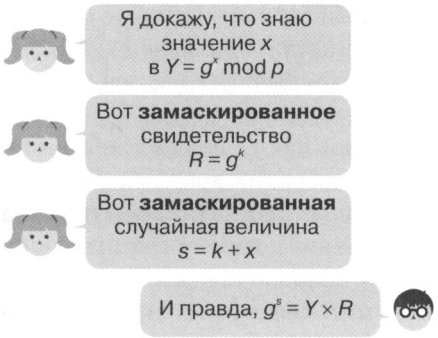


Рис. 7.5. Чтобы создать доказательство с нулевым разглашением, проверяющий маскирует свидетельство x с помощью случайной величины k, а затем маскирует саму случайную величину



Рис. 7.6. Протокол идентификации Шнорра — это интерактивный протокол ZKP, обладающий такими свойствами, как полнота (Пегги может доказать, что знает свидетельство), корректность (Пегги не может ничего доказать, если не знает свидетельства) и нулевое разглашение (Виктор ничего не может узнать о доказательстве)

Интерактивные системы ZKP, которые работают по схеме трех действий (обязательство, вызов и доказательство), в литературе часто называют *сигма-протоколами* (sigma protocols) и иногда обозначают как Σ-протоколы (с помощью греческой буквы). Но какое же отношение они имеют к цифровым подписям?

ПРИМЕЧАНИЕ

Протокол идентификации Шнорра работает по *модели нулевого разглашения с честным проверяющим* (honest verifier zero-knowledge model, HVZK): если проверяющий (Виктор) действует нечестно и вызов был выбран не случайно, он может узнать какую-нибудь информацию о свидетельстве. Некоторые более сильные схемы ZKP обладают нулевым разглашением, даже если проверяющий — злоумышленник.

7.2.2. Подписи как неинтерактивные доказательства с нулевым разглашением

Проблема с интерактивным протоколом ZKP заключается в его интерактивности, а в реальном мире такие протоколы, как правило, не любят. Интерактивные протоколы значительно повышают затраты ресурсов из-за необходимости обработки некоторого количества сообщений (предположительно, по сети) и делают *задержки неограниченными*, если только оба участника не находятся в сети одновременно. В связи с этим в мире прикладной криптографии интерактивные протоколы ZKP почти не используются.

Тем не менее обо всем этом было рассказано не просто так! В 1986 году Амос Фиат и Ади Шамир опубликовали методику, позволяющую легко преобразовать интерактивный протокол ZKP в его неинтерактивный вариант. Их трюк (так называемый *эвристический подход Фиата — Шамира*, или *преобразование Фиата — Шамира*) заключается в том, чтобы доказывающий сам вычислял вызов, причем таким образом, который он не может контролировать.

Вот в чем хитрость: необходимо вычислить вызов как хеш всех сообщений, отправленных и полученных в рамках протокола в известных пределах. Если мы предположим, что хеш-функция выдает выходные данные, неотличимые от истинно случайных чисел (другими словами, выглядит случайной), то она может успешно имитировать роль проверяющего.

Шнорр пошел еще дальше. Он заметил, что в этот хеш можно включить что угодно! Что, если добавить в него сообщение? Мы получим не только доказательство того, что нам известно некоторое свидетельство x, но и обязательство по сообщению, имеющее криптографическую связь с доказательством. Другими словами, если доказательство верно, то только тот, кто знает свидетельство (которое становится ключом подписи), мог зафиксировать это сообщение как обязательство.

Это и есть подпись! Цифровые подписи — это просто неинтерактивные протоколы ZKP. Применив преобразование Фиата — Шамира к протоколу идентификации Шнорра, мы получим *схему подписи Шнорра* (рис. 7.7).

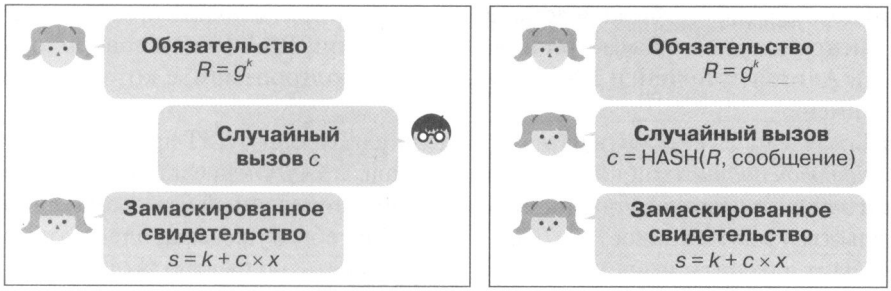


Рис. 7.7. *Слева* протокол идентификации Шнорра, о котором говорилось ранее, — интерактивный протокол. *Справа* протокол подписи Шнорра — его неинтерактивная версия, где сообщение проверяющего заменено хеш-функцией, вызванной для преобразования сообщений

Вкратце напомню, что подпись Шнорра — это, по сути, два значения, R и s, где R — обязательство по секретной случайной величине (которое часто называют *nonce*, поскольку оно должно быть уникальным для каждой подписи), s — значение, вычисляемое с помощью обязательства R, закрытого ключа (свидетельства x) и сообщения. Далее рассмотрим современные стандарты алгоритмов подписи.

7.3. АЛГОРИТМЫ ПОДПИСИ, КОТОРЫЕ СЛЕДУЕТ (ИЛИ НЕ СТОИТ) ИСПОЛЬЗОВАТЬ

Как и в других областях криптографии, для цифровой подписи существует множество стандартов, и иногда трудно понять, какой из них стоит выбрать. Именно поэтому я здесь! К счастью, алгоритмы подписи схожи с алгоритмами обмена ключами: есть основанные на арифметике по модулю большого числа, такие как алгоритмы Диффи — Хеллмана и RSA, и есть основанные на эллиптических кривых, например алгоритм Диффи — Хеллмана на эллиптических кривых (ECDH).

Убедитесь, что вы хорошо поняли, как работают эти алгоритмы, поскольку сейчас они послужат основой для представления стандартов подписи. Интересно, что в документе, в котором был представлен обмен ключами Диффи — Хеллмана, была предложена также концепция цифровых подписей (без указания решения).

> Для того чтобы разработать систему, способную заменить существующий письменный договор чисто электронной формой коммуникации, мы должны открыть цифровой феномен, обладающий теми же свойствами, что и рукописная подпись. Любой человек должен без труда признать подпись подлинной, но никто, кроме законного автора, не сможет ее поставить. Мы назовем любую такую технику односторонней аутентификацией. Поскольку любой цифровой сигнал может быть в точности воспроизведен, настоящая цифровая подпись должна быть узнаваема, не будучи известной.
>
> *Диффи и Хеллман («Новые направления в криптографии», 1976)*

Годом позже, в 1977-м, был представлен первый алгоритм подписи, названный RSA, а также алгоритм асимметричного шифрования RSA (о нем говорилось в главе 6). Алгоритм подписи RSA и будет первым алгоритмом, с которым мы познакомимся.

В 1991 году в попытке обойти патенты на подписи Шнорра NIST предложил *алгоритм цифровой подписи* (digital signature algorithm, DSA). Он представляет собой странный вариант протокола подписи Шнорра, опубликованный без какого-либо доказательства криптостойкости (хотя до сих пор не обнаружено ни одной атаки на него). Этот алгоритм имел некоторую популярность, но быстро был заменен версией на эллиптических кривых под названием *ECDSA* (Elliptic Curve Digital Signature Algorithm), то есть повторил судьбу алгоритма Диффи — Хеллмана, который также заменили из-за большого размера ключей (см. главу 5). ECDSA — это второй алгоритм подписи, о котором я расскажу в этом разделе.

После того как в 2008 году истек срок действия патентов на подписи Шнорра, Дэниел Дж. Бернштейн, изобретатель ChaCha20Poly1305 (см. главу 4) и X25519 (см. главу 5), использовал их для разработки новой схемы подписи *EdDSA* (Edwards-curve digital signature algorithm). EdDSA быстро завоевала признание и сейчас считается самой современной цифровой подписью, подходящей для применения в реальных приложениях. EdDSA — это третий, и последний, алгоритм подписи из этого раздела.

7.3.1. Плохой стандарт RSA PKCS#1 v1.5

Сейчас наиболее широко применяются подписи RSA, несмотря на то что с ними, как вы увидите в этом разделе, связано множество проблем. Такая популярность обусловлена тем, что данный алгоритм стал первой стандартизированной схемой подписи. К тому же реальные приложения не торопятся переходить на новые, более совершенные варианты. Поэтому, скорее всего, на своем профессиональном пути вы не раз столкнетесь с подписями RSA, и я не могу обойтись без объяснения их работы и связанных с ними стандартов. Впрочем, стоит отметить, что если вы поняли, как работает RSA-шифрование (глава 6), то данный раздел не должен вызвать у вас сложностей, потому что подписание по схеме RSA противоположно шифрованию с помощью RSA (рис. 7.8).

- Для того чтобы подписать сообщение, вы *шифруете* его с помощью закрытого ключа (а не открытого), в результате чего получается подпись (случайный элемент в группе).
- Для проверки подписи вы *расшифровываете* ее с помощью открытого ключа (а не закрытого). Если в результате получилось исходное сообщение, значит, подпись действительна.

ПРИМЕЧАНИЕ

В реальности перед подписанием сообщение, как правило, хешируется, поскольку в таком виде оно занимает меньше места (с помощью RSA можно подписывать только сообщения, размер которых меньше его модуля). Результат также интерпретируется как большое число, чтобы его можно было применять в математических операциях.

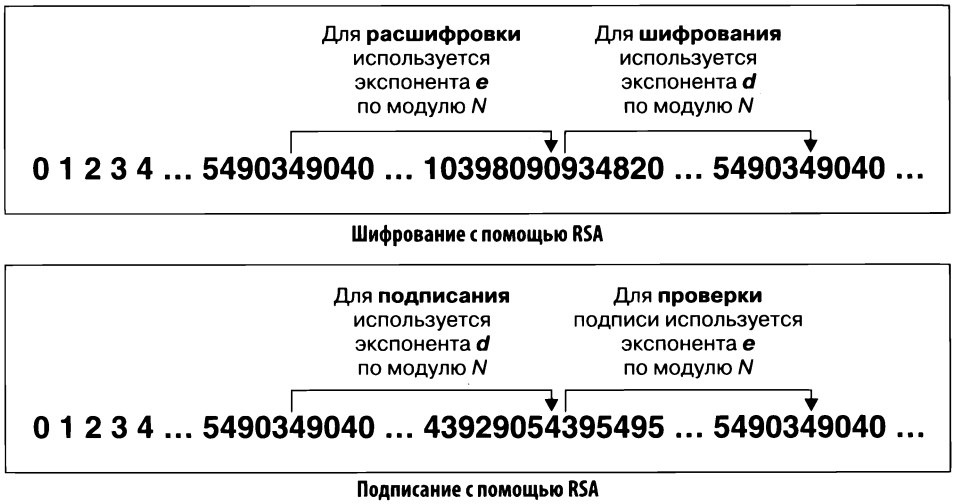


Рис. 7.8. Для подписания с помощью RSA мы просто выполняем алгоритм шифрования RSA в обратном порядке: возводим сообщение в секретную степень, затем для проверки подписи возводим ее в открытую степень, результатом чего должно быть исходное сообщение

Если закрытый ключ — это секретная экспонента d, а открытый ключ — открытая экспонента e и открытый модуль N, то можно:

- подписать сообщение, вычислив $подпись = сообщение^d \bmod N$;
- проверить подпись на ее соответствие сообщению, вычислив $подпись^e \bmod N$.

Эти действия наглядно представлены на рис. 7.8.

Получается, что только тот, кто знает секретную экспоненту d, может подписать сообщение. Как и в случае с RSA-шифрованием, уровень криптостойкости этого алгоритма подписи тесно связан со сложностью задачи факторизации.

А что насчет стандартов, использующих RSA для подписания? К счастью, здесь все так же, как и с RSA-шифрованием.

- *В документе PKCS#1 v1.5 был представлен неудачный стандарт алгоритма RSA для шифрования.* В нем же содержалась спецификация для подписания с помощью RSA (без доказательства криптостойкости).
- *Затем RSA был снова стандартизирован в документе PKCS#1 v2, где представлена более совершенная конструкция (RSA-OAEP).* То же самое произошло и с RSA-подписями: алгоритм RSA-PSS был стандартизирован в том же документе (с доказательством криптостойкости).

Я рассказывал об RSA PKCS#1 v1.5 в главе 6, посвященной асимметричному шифрованию. Схема подписи, стандартизированная в этом документе, практически не отличается от схемы шифрования. Перед подписанием сообщение хешируется с помощью выбранной хеш-функции, а затем дополняется в соответствии с правилами дополнения подписей PKCS#1 v1.5, аналогичными правилам дополнения

при шифровании, изложенным в том же стандарте. Затем дополненное и хешированное сообщение шифруется с помощью секретной экспоненты (рис. 7.9).

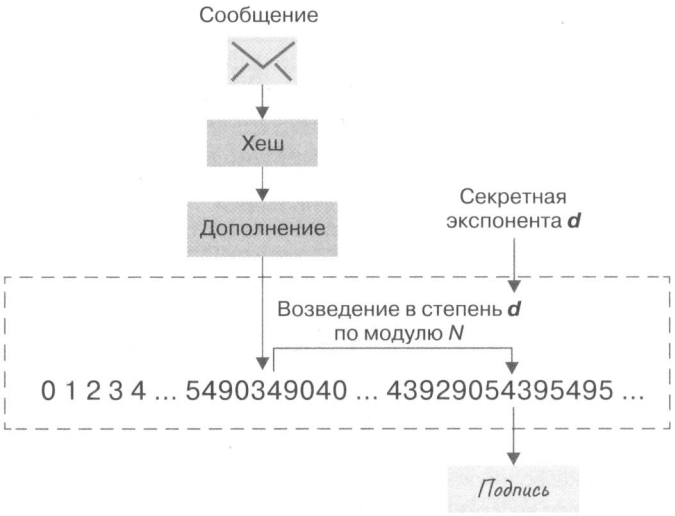


Рис. 7.9. RSA PKCS#1 v1.5 для подписей. Для того чтобы подписать сообщение, его хешируют, а затем дополняют по схеме дополнения PKCS#1 v1.5. На конечном этапе закрытый ключ d по модулю N выступает в качестве степени, в которую возводят дополненное и хешированное сообщение. Для проверки подписи достаточно возвести подпись в открытую степень e по модулю N и убедиться, что она соответствует дополненному и хешированному сообщению

БОЛЬШОЕ КОЛИЧЕСТВО RSA

Термины, связанные с RSA, требуют особого внимания: RSA может обозначать как *примитив асимметричного шифрования*, так и *примитив подписи*. Кроме того, есть еще и компания под названием RSA, основанная изобретателями RSA. При упоминании шифрования с помощью RSA в большинстве случаев имеются в виду схемы RSA PKCS#1 v1.5 и RSA-OAEP, при упоминании RSA-подписей — схемы RSA PKCS#1 v1.5 и RSA-PSS.

Я знаю, что особую путаницу вызывает стандарт PKCS#1 v1.5. Хотя в нем есть официальные названия для различения алгоритмов (RSAES-PKCS1-v1_5 для шифрования и RSASSA-PKCS1-v1_5 для подписи), я редко вижу, чтобы они использовались.

В главе 6 я затрагивал тему разрушительных атак на алгоритм шифрования RSA PKCS#1 v1.5. К сожалению, это относится и к подписям RSA PKCS#1 v1.5. В 1998 году, после обнаружения атаки миллионом сообщений, Блейхенбахер решил взглянуть на стандарт подписей. Он вернулся в 2006-м с атакой на RSA PKCS#1 v1.5 через *подделку подписи* — одним из самых катастрофических типов атак на

подписи: атакующие могут подделывать их без знания закрытого ключа! В отличие от первой атаки, которая взламывала непосредственно алгоритм шифрования, данная атака затрагивала реализацию. Следовательно, если схема подписи реализована правильно (в соответствии со спецификацией), атака не сработает.

Уязвимость реализации звучит не так плохо, как уязвимость алгоритма: ее легко избежать, и она не влияет на многие другие реализации. К сожалению, в 2019 году одна статья показала, что досадно большое количество реализаций RSA PKCS#1 v1.5 для подписей с открытым исходным кодом попались на удочку и оказались уязвимыми (см. *Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification*, автор Чо и др.). В итоге ряд уязвимостей реализации позволили появиться различным вариантам атаки через подделку Блейхенбахера.

К сожалению, стандарт RSA PKCS#1 v1.5 для подписей все еще широко применяется. Помните о связанных с ним проблемах, если вам действительно *необходимо* задействовать этот алгоритм по соображениям обратной совместимости. Однако нельзя сказать, что RSA для подписей вообще небезопасен. На этом его история не заканчивается.

7.3.2. Лучший стандарт RSA-PSS

Алгоритм RSA-PSS был стандартизирован в обновленной версии PKCS#1 v2.1 и имел доказательство криптостойкости, в отличие от схемы подписи, стандартизированной в PKCS#1 v1.5. Новая спецификация работает следующим образом:

- сообщение кодируется посредством алгоритма кодирования PSS;
- закодированное сообщение подписывается с помощью RSA (как в стандарте PKCS#1 v1.5).

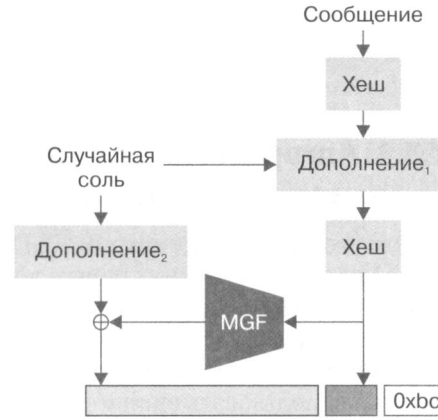


Рис. 7.10. Схема подписи RSA-PSS, подобно алгоритму RSA-OAEP из главы 6, кодирует сообщение с помощью функции генерации масок (MGF) и только потом добавляет к нему обычную RSA-подпись

Кодирование PSS похоже на OAEP (оптимальное асимметричное шифрование с дополнением). Принцип его работы проиллюстрирован на рис. 7.10.

Проверка подписи, полученной с помощью RSA-PSS, сводится к инвертированию кодировки после возведения подписи в открытую степень по открытому модулю.

Если помните, в главе 6 был представлен третий алгоритм шифрования RSA: он называется RSA-KEM, имеет упрощенную структуру и доказанную криптографическую криптостойкость, однако нигде не применяется. Интересно, что RSA для подписей также отражает эту часть истории шифрования RSA: существует более простой алгоритм, который практически никто не использует, под названием Full

Domain Hash (FDH). Он действует путем простого хеширования и последующего подписания сообщения (интерпретируя дайджест как число) с помощью RSA.

ДОКАЗУЕМАЯ КРИПТОСТОЙКОСТЬ PSS

Вероятностная схема подписи (Probabilistic Signature Scheme, PSS) является доказуемо криптостойкой. Таким образом, никто не сможет подделать подпись, не зная закрытого ключа. Доказательство криптостойкости RSA-PSS строится не на доказательстве криптостойкости RSA, а на обратном: если кто-то сможет взломать RSA-PSS, то он может взломать и RSA. Это обычный способ доказательства в криптографии. Конечно, он работает только в том случае, если RSA безопасен, что и предполагается в доказательстве.

Несмотря на то что криптостойкость RSA-PSS и FDH доказана и они гораздо проще в реализации, чем RSA PKCS#1 v1.5, сегодня большинство протоколов по-прежнему применяют последний для подписей. Это еще один пример медлительности, которая возникает при использовании устаревающих криптографических алгоритмов. Поскольку старые реализации все равно должны быть совместимы с новыми, удалить или заменить алгоритмы сложно. Достаточно вспомнить пользователей, которые не обновляют приложения, производителей, которые не предоставляют новые версии своего программного обеспечения, или аппаратные устройства, поддержка которых уже закончилась. Далее поговорим о более современном алгоритме.

7.3.3. Алгоритм цифровой подписи на эллиптических кривых

В этом разделе мы рассмотрим ECDSA — вариант алгоритма цифровой подписи (DSA) на эллиптических кривых, который был изобретен только для того, чтобы обойти патенты на подписи Шнорра. Данный алгоритм описан во многих стандартах, включая ISO 14888-3, ANSI X9.62, NIST's FIPS 186-2, IEEE P1363 и др. Не все стандарты совместимы, поэтому взаимодействующие друг с другом приложения должны использовать один и тот же стандарт.

К сожалению, в отличие от схемы подписи Шнорра, ни ECDSA, ни DSA не имеют доказательств криптостойкости. Тем не менее ECDSA получил широкое распространение и является одной из самых часто применяемых схем подписи. В этом разделе я объясню, как он работает и как его можно использовать. Как и во всех подобных схемах, открытый ключ в ECDSA практически всегда генерируется по одной и той же формуле:

- закрытый ключ — это сгенерированное случайным образом большое число x;
- открытый ключ получается, если рассматривать x как индекс в группе, созданной образующим элементом (в криптографии с эллиптическими кривыми он называется *базовой точкой*).

Если говорить конкретнее, в ECDSA открытый ключ вычисляется выражением $[x]G$, которое представляет собой произведение скаляра x и базовой точки G.

АДДИТИВНАЯ ИЛИ МУЛЬТИПЛИКАТИВНАЯ ФОРМА ЗАПИСИ?

Обратите внимание на то, что я использую *аддитивную форму записи* (с синтаксисом эллиптической кривой, заключающим скаляр в квадратные скобки), но если бы хотел применять *мультипликативную запись*, то мог бы представить это выражение как *закрытый_ключ* = G^x. На практике это не имеет значения: в большинстве случаев криптографические протоколы, которым неважна природа группы, составлены с использованием мультипликативной формы записи, а протоколы, определенные специально для групп на основе эллиптических кривых, — аддитивной.

Для вычисления подписи ECDSA понадобятся те же исходные данные, что и для подписи Шнорра: хеш подписываемого сообщения $H(m)$, закрытый ключ x и уникальное для каждой подписи случайное число k. Подпись ECDSA состоит из двух целых чисел, r и s, которые вычисляются следующим образом:

- r — это координата на оси X кривой $[k]G$;
- $s = k^{-1}(H(m) + xr) \bmod p$.

Чтобы проверить подпись ECDSA, проверяющий должен использовать то же самое хешированное сообщение $H(m)$, открытый ключ подписывающего и значения подписи r и s. Затем проверяющий делает следующее.

1. Вычисляет $[H(m)s^{-1}]G + [rs^{-1}]$ *закрытый_ключ*.
2. Проверяет, чтобы координата на оси X полученной точки совпадала со значением r в подписи.

Вы, конечно, можете заметить, что здесь есть некоторое сходство с подписями Шнорра. Случайное число k иногда называют nonce, поскольку оно должно быть использовано только один раз, а также *эфемерным ключом*, так как оно должно оставаться в секрете.

ВНИМАНИЕ

Скажу еще раз: число k не должно повторяться и быть предсказуемым! Если это условие не соблюдается, восстановить закрытый ключ проще простого.

Как правило, криптографические библиотеки выполняют генерацию такого nonce (величины k) скрытно, но иногда они этого не делают и перекладывают данное действие на вызывающую сторону. Конечно, это верный путь к беде. Например, в 2010 году было обнаружено, что Sony Playstation 3 использует ECDSA с повторяющимися nonce, что привело к утечке закрытых ключей.

ВНИМАНИЕ

Есть и еще более тонкий момент: если выбор nonce k носил неравномерный и неслучайный характер (особенно если вы можете предсказать первые несколько битов), существуют мощные атаки, которые могут восстановить закрытый ключ в мгновение ока *(атаки на решетку)*. В теории такие атаки по восстановлению ключа называются *полными взломами* (потому что они ломают все!). На практике полные взломы встречаются довольно редко, что делает ECDSA тем алгоритмом, взлом которого наделает много шума.

Проблем с nonce пытались избежать несколькими способами. Например, RFC 6979 определяет *детерминированную* схему ECDSA, генерирующую nonce на основе сообщения и закрытого ключа: при подписании одного и того же сообщения дважды задействован один и тот же *nonce* и, следовательно, дважды создается одна и та же подпись (что, безусловно, не проблема).

Эллиптические кривые, которые обычно используются в ECDSA, — это практически те же кривые алгоритма ECDH (см. главу 5), за одним заметным исключением — *Secp256k1*. Кривая Secp256k1 была определена в SEC 2: Recommended Elliptic Curve Domain Parameters (https://secg.org/sec2-v2.pdf), написанном группой Standards for Efficient Cryptography (SECG). Она получила широкое распространение после того, как биткойн решил использовать ее вместо более популярных, но утративших доверие кривых от NIST, о которых я упоминал в главе 5.

Secp256k1 относится к типу эллиптических кривых, называемых *кривыми Коблица*. Кривая Коблица — это просто эллиптическая кривая с некоторыми ограничениями по параметрам, которые позволяют реализации оптимизировать некоторые операции над ней. Уравнение эллиптической кривой:

$$y^2 = x^3 + ax + b,$$

где $a = 0$ и $b = 7$ — константы, а x и y определены над числами по модулю простого числа p:

$$p = 2^{192} - 2^{32} - 2^{12} - 2^8 - 2^7 - 2^6 - 2^3 - 1.$$

Данное уравнение, подобно кривым от NIST, определяет группу простого порядка. Сегодня у нас есть эффективные формулы для вычисления количества точек на эллиптической кривой. Вот простое число, равное количеству точек на кривой Secp256k1 (включая точку на бесконечности): 115 792 089 237 316 195 423 570 985 008 687 907 852 837 564 279 074 904 382 605 163 141 518 161 494 337.

В качестве образующего элемента (или базовой точки) мы используем фиксированную точку G с координатами x = 55 066 263 022 277 343 669 578 718 895 168 534 326 250 603 453 777 594 175 500 187 360 389 116 729 240 и y = 32 670 510 020 758 816 978 083 085 130 507 043 184 471 273 380 659 243 275 938 904 335 757 337 482 424.

Тем не менее сегодня ECDSA в основном используется с кривой NIST P-256, которую иногда называют *Secp256r1* — обратите внимание на эту разницу. Далее рассмотрим еще одну широко распространенную схему подписи.

7.3.4. Алгоритм цифровой подписи на кривой Эдвардса

Последним представленным в этой главе алгоритмом подписи является *алгоритм цифровой подписи на кривой Эдвардса* (Edwards-Curve Digital Signature Algorithm, EdDSA), опубликованный в 2011 году Дэниелом Дж. Бернштейном в качестве замены потерявшим доверие кривым от NIST и правительственных агентств. Из названия может показаться, что EdDSA основан на алгоритме DSA, как и ECDSA, но это не так. На самом деле EdDSA базируется на подписях Шнорра, срок действия патента на которые истек в начале 2008 года.

Особенностью EdDSA является то, что он не требует новой случайности для каждой операции подписания. *Детерминированное* создание подписей сделало этот алгоритм весьма привлекательным, и впоследствии он был принят во многих протоколах и стандартах.

EdDSA находится в стадии подготовки к включению в предстоящее обновление стандарта NIST FIPS 186-5 (по состоянию на начало 2021 года он еще разрабатывается). Действующим официальным стандартом является RFC 8032, в котором определены две кривые разного уровня безопасности для использования с EdDSA. Обе они являются кривыми Эдвардса (типом эллиптической кривой, позволяющим интересные оптимизации реализации).

- *Кривая Edwards25519 основана на кривой Curve25519 Дэниела Дж. Бернштейна* (см. главу 5). Благодаря оптимизациям, предоставляемым этим типом эллиптической кривой, Edwards25519 отличается более быстрой реализацией операций на кривых. Поскольку данная кривая была изобретена уже после Curve25519, обмен ключами X25519, в основе которого лежит Curve25519, не выиграл от увеличения скорости. Как и Curve25519, Edwards25519 обеспечивает 128-битную защиту.
- *Кривая Edwards448 основана на кривой Ed448-Goldilocks Майка Гамбурга*. Она обеспечивает 224-битную защиту.

На практике алгоритм EdDSA чаще всего используется с кривой Edwards25519, их сочетание носит название *Ed25519* (а EdDSA и кривая Edwards448 — Ed448). Генерация ключей в EdDSA немного отличается от других схем: вместо того чтобы сразу создать ключ подписи, алгоритм генерирует секретный ключ, который затем применяется для получения текущего ключа подписи и nonce-ключа. Nonce-ключ очень важен! Именно он используется для детерминированности генерации, требующейся для каждой подписи.

ПРИМЕЧАНИЕ

В зависимости от применяемой криптографической библиотеки вы можете хранить секретный ключ или два производных ключа: ключ подписи и nonce-ключ. Это не самая важная особенность алгоритма, но если о ней не знать, секретные ключи Ed25519 размером 32 или 64 байта, в зависимости от используемой реализации, могут сбить с толку.

Для создания подписи EdDSA сначала детерминированно генерирует nonce с помощью хеширования nonce-ключа с сообщением, которое нужно подписать. После этого идет процесс, аналогичный схеме подписи Шнорра.

1. Вычисляется nonce как *HASH* (*nonce-ключ* || *сообщение*).
2. Вычисляется обязательство R как $[nonce]G$, где G — базовая точка группы.
3. Вычисляется вызов как *HASH* (*обязательство* || *открытый ключ* || *сообщение*).
4. Вычисляется доказательство S как *nonce* + *вызов* × *ключ подписи*.

Подпись — это (R, S). Я иллюстрирую важные части EdDSA на рис. 7.11.

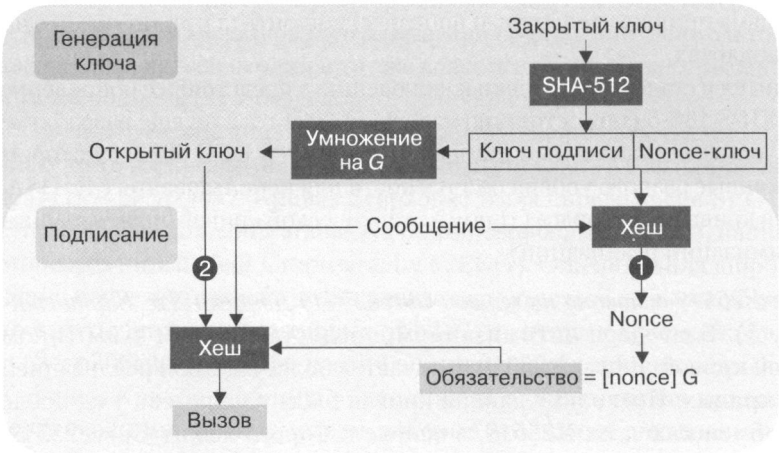


Рис. 7.11. При генерации ключа EdDSA создается секретный ключ, с помощью которого затем получают два других ключа. Первый производный ключ — это текущий ключ подписи, он может быть использован для получения открытого ключа. Другой производный ключ — это nonce-ключ для детерминированного получения nonce во время подписания. Подписи EdDSA похожи на подписи Шнорра, за исключением того, что nonce генерируется детерминированно из nonce-ключа и сообщения *(1)*, а открытый ключ подписывающего включен в вызов *(2)*

Обратите внимание на детерминированный и упорядоченный характер генерации nonce (или эфемерного ключа) на основе nonce-ключа и заданного сообщения. Для подписания двух разных сообщений должны использоваться два разных nonce, что позволяет предотвратить повторное применение nonce и, соответственно, утечку ключа, как может произойти в ECDSA. Подписывая одно и то же сообщение дважды, вы дважды получаете один и тот же nonce, который затем дважды создает одну и ту же подпись. Конечно, это не проблема. Подпись можно проверить, вычислив следующие два уравнения:

$$[S]G;$$

$$R + [\textit{хеш}(R \parallel \textit{закрытый ключ} \parallel \textit{сообщение})]\ \textit{открытый ключ}.$$

Подпись считается действительной, если эти два значения совпадают. Именно так работают подписи Шнорра, за исключением того, что теперь мы находимся в группе эллиптических кривых, а значит, используем аддитивную форму записи.

Наиболее распространенный вариант EdDSA — Ed25519 определяется кривой Edwards25519 и хеш-функцией SHA-512. Кривая Edwards25519 определяется всеми точками, удовлетворяющими уравнению:

$$-x^2 + y^2 = 1 + dx^2y^2 \bmod p,$$

где значение d — большое число 37 095 705 934 669 439 343 138 083 508 754 565 189 542 113 879 843 219 016 388 785 533 085 940 283 555, а переменные x и y берутся по

модулю p большого числа $2^{255} - 19$ (того же простого числа, которое используется для Curve25519). Базовой точкой является G с координатами x = 15 112 221 349 535 400 772 501 151 409 588 531 511 454 012 693 041 857 206 046 113 283 949 847 762 202 и y = 46 316 835 694 926 478 169 428 394 003 475 163 141 307 993 866 256 225 615 783 033 603 165 251 855 960.

RFC 8032 фактически определяет три варианта алгоритма EdDSA, использующих кривую Edwards25519. Во всех них один и тот же алгоритм генерации ключей, но разные алгоритмы подписи и проверки:

- *Ed25519* (или *pureEd25519*) — алгоритм, о котором говорилось ранее;
- *Ed25519ctx* — вводит обязательную строку настройки и на практике применяется редко, если вообще применяется. От предыдущего алгоритма отличается лишь тем, что к каждому вызову хеш-функции добавляется выбранный пользователем префикс;
- *Ed25519ph* (или *HashEd25519*) — позволяет приложениям хешировать сообщение перед его подписанием (отсюда *ph* в названии). Он основан на *Ed25519ctx*, что позволяет вызывающей стороне добавить опциональную строку настройки.

Как можно увидеть на примере некоторых хеш-функций из главы 2 и функций формирования ключей из главы 8, добавление *строки настройки* в криптографии встречается довольно часто. Полезно, когда участник протокола в разных случаях задействует один и тот же ключ для подписания сообщений. В качестве примера можно взять приложение, позволяющее пользователю с помощью своего закрытого ключа подписывать транзакции и личные сообщения людям. Если он по ошибке подпишет и отправит сообщение, похожее на транзакцию, своей злобной подруге Еве, она может попытаться опубликовать его как настоящую транзакцию, если не будет возможности различить два типа подписываемых наборов данных.

Алгоритм Ed25519ph был введен исключительно для тех пользователей, которым нужно подписывать большие сообщения. Как вы видели в главе 2, хеш-функции часто предоставляют систему init — update — finalize, которая позволяет непрерывно хешировать поток данных, не храня все входные данные в памяти.

На этом можно и закончить представление схем подписи, которые используются в реальных приложениях. Далее рассмотрим, как можно выстрелить себе в ногу, если неаккуратно обращаться с ними. Но сначала напомню.

- Алгоритм RSA PKCS#1 v1.5 все еще широко распространен, но его трудно правильно реализовать, и многие реализации были признаны нерабочими.
- RSA-PSS имеет доказательство криптостойкости, проще в реализации, но плохо внедряется из-за более новых схем, основанных на эллиптических кривых.
- ECDSA — главный конкурент RSA PKCS#1 v1.5 и в основном используется с кривой P-256 от NIST, за исключением мира криптовалюты, где по всем признакам доминирует Secp256k1.
- Широкое распространение получил основанный на подписях Шнорра алгоритм Ed25519. Его также проще реализовать по сравнению с ECDSA, и он не требует новой случайности для каждой операции подписи. По возможности следует использовать именно его.

7.4. ТОНКОСТИ СХЕМ ПОДПИСИ

У схем подписи может проявиться ряд коварных свойств. Хотя в большинстве случаев они ни на что не влияют, незнание этих тонкостей способно помешать в ходе работы над сложными и необычными протоколами. В конце главы мы рассмотрим известные проблемы, связанные с цифровыми подписями.

7.4.1. Атаки на подписи через подмену

> Цифровая подпись не позволяет однозначно идентифицировать ключ или сообщение.
>
> *Эндрю Айер (Duplicate Signature Key Selection Attack in Let's Encrypt, 2015)*

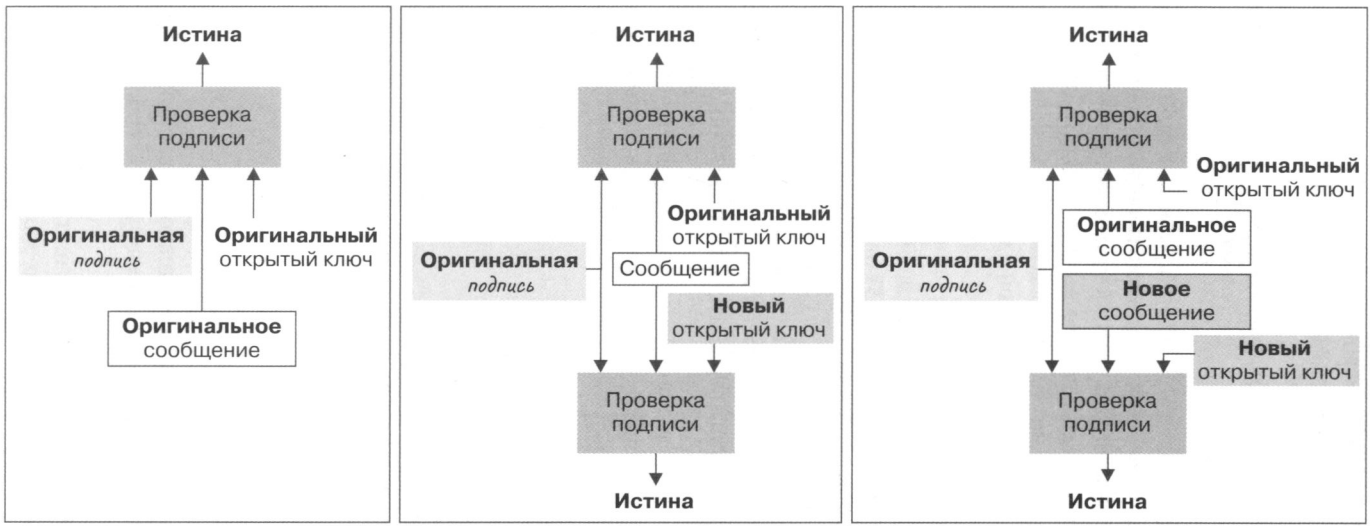
Атаки через подмену, известные также как *подбор дубликата ключа подписи* (duplicate signature key selection, DSKS), возможны как для RSA PKCS#1 v1.5, так и для RSA-PSS. Существует два варианта DSKS.

- *Атаки через подмену ключа* — для проверки конкретной подписи под конкретным сообщением используется другая пара ключей или другой открытый ключ.
- *Атаки через подмену ключа сообщения* — для проверки конкретной подписи под *новым* сообщением применяется другая пара ключей или другой открытый ключ.

Еще раз: первая атака заменяет как сообщение, так и подпись, вторая — только подпись (рис. 7.12).

СТОЙКОСТЬ К ЭКЗИСТЕНЦИАЛЬНОЙ ПОДДЕЛКЕ ОТНОСИТЕЛЬНО АТАК НА ОСНОВЕ АДАПТИВНО ПОДОБРАННЫХ СООБЩЕНИЙ

Атаки через подмену являются симптомами разрыва между теоретическими и прикладными областями криптографии. Анализ криптографической подписи обычно выполняется с помощью *модели EUF-CMA*, название которой можно расшифровать как стойкость к экзистенциальной подделке относительно атак на основе адаптивно подобранных сообщений (existential unforgeability under adaptive chosen message attack). В этой модели один человек генерирует пару ключей, а затем его просят подписать несколько произвольных сообщений. Другой человек, наблюдающий за создаваемыми подписями, выигрывает, если в какой-то момент ему удается создать действительную подпись под сообщением, которого он не запрашивал. К сожалению, вероятнее всего, EUF-CMA не охватывает каждый пограничный случай и не принимает во внимание такие опасные моменты, как подмена.



Нормальная подпись **Атака через подмену ключа** **Атака через подмену ключа сообщения**

Рис. 7.12. Алгоритмы подписи, например RSA, уязвимы к атакам через подмену ключа. Такие атаки застают врасплох большую часть не ожидавших подвоха пользователей криптографии. Атака через подмену ключа позволяет взять подпись под сообщением и создать новую пару ключей, подтверждающую оригинальную подпись. Вариант с подменой ключа сообщения позволяет злоумышленнику создать новую пару ключей и новое сообщение, подтверждающие оригинальную подпись

7.4.2. Деформируемость подписи

> В феврале 2014 года некогда крупнейшая биткойн-биржа MtGox объявила о банкротстве и закрылась, обвинив злоумышленников в атаке через деформируемость, результатом которой стала утечка средств с ее счетов.
>
> *Кристиан Декер и Роджер Уоттенхофер (Bitcoin Transaction Malleability and MtGox, 2014)*

Большинство схем подписи являются *деформируемыми*: если злоумышленнику дать действительную подпись, он сможет изменить ее до неузнаваемости и при этом она останется действительной. Злоумышленник понятия не имеет, каким был ключ подписи, но ему удалось создать новую действительную подпись.

Недеформируемость подписи не обязательно подразумевает ее уникальность: подписывающий может создать разные подписи для одного и того же сообщения, и это, как правило, не вызывает проблем. В некоторых конструкциях, таких как, например, проверяемые функции генерации случайных чисел (о которых вы узнаете в главе 8), уникальность подписи играет очень важную роль, поэтому они должны либо сами обеспечить ее, либо использовать какую-нибудь схему уникальных подписей, например подписи Боне – Линна – Шахама (BLS).

Что делать со всей этой информацией? Будьте уверены, схемы подписи работают и при их стандартном применении не о чем беспокоиться. Однако, если вы разрабатываете криптографические протоколы или реализуете протокол, выходящий за рамки повседневной криптографии, вам, возможно, не стоит забывать об этих тонкостях.

СИЛЬНАЯ EUF-CMA

Новая модель обеспечения безопасности SUF-CMA (от strong EUF-CMA) пытается включить недеформируемость (или устойчивость к деформациям) в определение криптоустойчивости схем подписи. В некоторые последние стандарты, например RFC 8032, определяющий Ed25519, уже включены средства защиты от атак через деформируемость подписей. Из-за того что эти средства защиты слабо распространены и не всегда доступны, необходимо подразумевать деформируемость подписей.

РЕЗЮМЕ

- Цифровые подписи похожи на рукописные, однако криптографическая природа защищает их от подделки теми, кто не контролирует ключ подписи (закрытый ключ).
- Цифровые подписи могут быть полезны для аутентификации источника (например, одной из сторон обмена ключами), а также для обеспечения транзитивного доверия (если я доверяю Алисе, а она доверяет Бобу, я могу доверять Бобу).

- Доказательства с нулевым разглашением (ZKP) позволяют человеку доказать, что он обладает определенной информацией (называемой свидетельством), не раскрывая ее. Подписи можно рассматривать как неинтерактивные схемы ZKP, поскольку не требуется, чтобы проверяющий находился в сети во время подписания.
- Подписание определено множеством стандартов.
  - На сегодняшний день алгоритм RSA PKCS#1 v1.5 широко применяется, но не рекомендуется, так как его трудно правильно реализовать.
  - RSA-PSS – лучшая схема подписи, поскольку она проще в реализации и имеет доказательство криптостойкости. К сожалению, сейчас она не пользуется популярностью из-за вариантов схем на основе эллиптических кривых, которые поддерживают более короткие ключи и, следовательно, более привлекательны для сетевых протоколов.
  - Наиболее популярные в настоящее время схемы подписи основаны на эллиптических кривых: ECDSA и EdDSA. ECDSA часто используется с кривой P-256, разработанной NIST, а EdDSA – с кривой Edwards25519 (эту комбинацию называют Ed25519).
- Подписи имеют некоторые тонкости, которые могут представлять опасность при нестандартном способе применения.
  - Избегайте двусмысленности в вопросе о том, кто подписал сообщение, поскольку некоторые схемы подписи уязвимы для атак через подмену ключа. Третьи лица могут создать новую пару ключей, которая подтвердит уже существующую подпись под сообщением, или создать новую пару ключей и новое сообщение, подтверждающие данную подпись.
  - Не полагайтесь на уникальность подписей. Во-первых, в большинстве схем подписи подписывающий может создать произвольное количество подписей для одного и того же сообщения. Во-вторых, большинство схем подписи *деформируемые*, то есть третьи лица могут взять подпись и на ее основе создать другую действительную подпись для того же сообщения.

8 Случайность и секреты

В этой главе

- ✓ Что такое случайность и почему она важна.
- ✓ Способы получения сильной случайности и формирование секретов.
- ✓ Ловушки случайности.

Вот и подходит к концу часть I. Прежде чем идти дальше и познакомить вас с настоящими протоколами из реального мира, я хотел бы затронуть последний элемент криптографии, которому до сих пор уделял незаслуженно мало внимания, — случайность.

Вы наверняка заметили, что в каждом рассмотренном нами криптографическом алгоритме, за исключением хеш-функций, в определенный момент приходилось использовать какую-нибудь случайность: секретные ключи, nonce, векторы инициализации, простые числа, вызовы. Она всегда появлялась из какого-то волшебного черного ящика. В этом нет ничего странного. В технических криптографических документах случайность часто обозначают стрелкой со знаком доллара наверху. Однако в какой-то момент мы должны были задать себе вопрос: «Откуда на самом деле она берется?»

В этой главе я дам ответ на вопрос, что именно подразумевает криптография под словом «случайность». А также расскажу о практических способах формирования случайности в реальных криптографических приложениях.

ПРИМЕЧАНИЕ

Чтобы лучше понять эту главу, рекомендуем прочитать главу 2, посвященную хеш-функциям, и главу 3, где рассматриваются имитовставки.

8.1. ЧТО ТАКОЕ СЛУЧАЙНОСТЬ

Каждому человеку в той или иной степени знакома концепция случайности. Мы все с ней встречались, играя в кости или покупая лотерейные билеты. Впервые я столкнулся со случайностью в очень раннем возрасте, когда увидел, что при каждом нажатии на кнопку RAND калькулятор выдавал разные числа. Это не давало мне покоя. Мало что зная об электронике, я все же думал, что смогу понять некоторые ее ограничения. Безусловно, когда я складывал 4 и 5, какие-то микросхемы выполняли вычисление и выдавали результат. Но что насчет кнопки случайности? Откуда взялись случайные числа? Я не мог взять этого в толк.

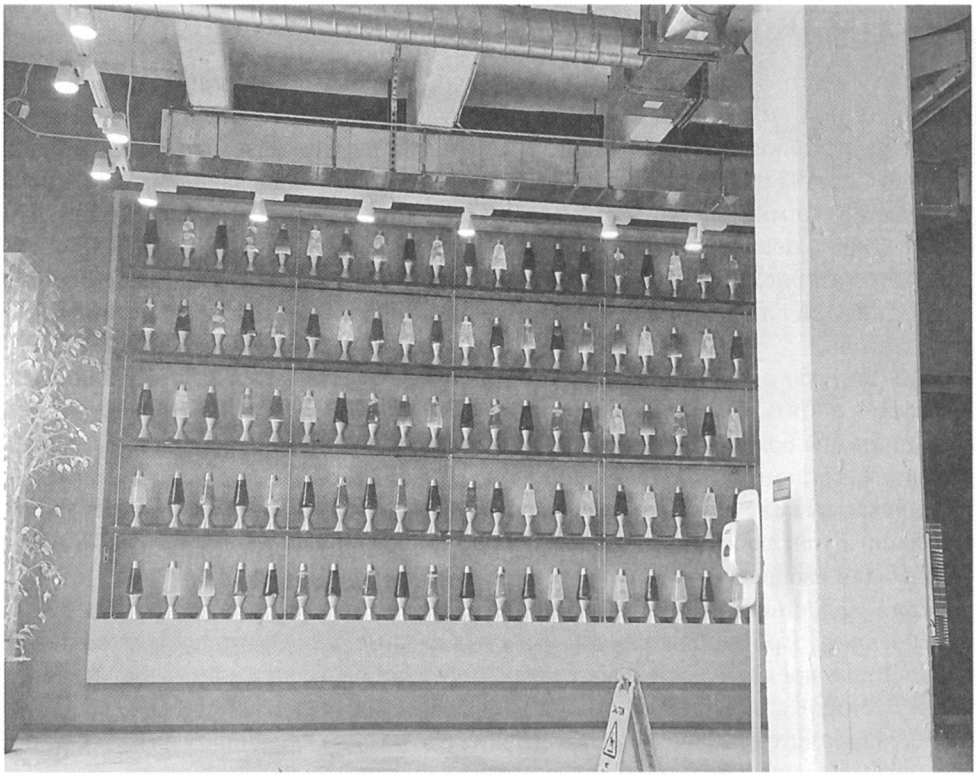
Мне потребовалось некоторое время для того, чтобы начать задавать правильные вопросы и понять, что калькулятор на самом деле обманывал! В нем были жестко закодированы большие списки случайных чисел, которые перебирались одно за другим. Они демонстрировали хорошую случайность, то есть если посмотреть на получаемые случайные числа, то среди них 1 будет встречаться так же часто, как 9, или 2, или любое другое число. Они имитировали *равномерное распределение*: числа были распределены в равных пропорциях (равномерно).

Если случайные числа нужны для целей, связанных с обеспечением безопасности и криптографией, то случайность должна иметь *непредсказуемый* характер. Конечно, во времена моей юности никто не стал бы использовать случайность калькуляторов для чего бы то ни было, связанного с обеспечением безопасности. Между тем криптографические приложения извлекают случайность из наблюдения за труднопредсказуемыми физическими явлениями.

Например, трудно предсказать результат бросания игральной кости даже с учетом того, что само по себе оно является детерминированным процессом. Если знать все начальные условия (характер броска, параметры кости, трение воздуха, сцепление со столом и т. д.), предугадать результат возможно. Тем не менее все эти факторы настолько сильно влияют на конечный результат, что даже небольшая неточность может привести к ошибке. Чрезвычайная чувствительность результата к начальным условиям известна как *теория хаоса*, и именно из-за нее, например, трудно точно предсказать погоду на большое количество дней.

Следующее изображение — фотография, сделанная мною во время одного из визитов в штаб-квартиру Cloudflare в Сан-Франциско. LavaRand — это стена из лавовых ламп, создающих труднопредсказуемые формы из воска. Перед стеной установлена камера, которая извлекает изображения и преобразует их в случайные байты.

В плане обеспечения практичной случайности приложения обычно полагаются на операционную систему (ОС), которая, в свою очередь, генерирует случайность с помощью различных трюков в зависимости от типа устройства, на котором она работает. Обычными источниками случайности (также называемыми *источниками энтропии*) могут быть фактическое время аппаратных прерываний (например, движения мыши), программные прерывания, время поиска данных на жестком диске и др.



ЭНТРОПИЯ

В теории информации слово *«энтропия»* используется для определения степени случайности в строке. Термин был придуман Клодом Шенноном, разработавшим формулу энтропии, которая выводила все большие и большие числа по мере увеличения непредсказуемости строки, начиная с 0 — полной предсказуемости. Сами по себе формула или число не играют для нас особой роли, тем не менее в криптографии часто можно услышать: «Эта строка имеет низкую энтропию» (она предсказуема) или «Эта строка имеет высокую энтропию» (она менее предсказуема).

Формирование случайности на основе наблюдений за прерываниями и другими событиями — не самое лучшее решение. При запуске устройства такие события, как правило, хорошо предсказуемы и подвержены злонамеренному воздействию со стороны внешних факторов. Сегодня все больше и больше устройств имеют доступ к дополнительным датчикам и аппаратным средствам, предоставляющим лучшие источники энтропии. Такие аппаратные генераторы случайных чисел часто называют *генераторами истинно случайных чисел* (true random number generators,

TRNG), поскольку для извлечения случайности они используют внешние непредсказуемые физические явления, например тепловой шум.

Как правило, шум, полученный с помощью всех этих типов входных данных, не является чистым и временами не обеспечивает достаточный уровень энтропии, если вообще обеспечивает. Например, первый бит, полученный из какого-либо источника энтропии, очень часто может быть равен 0 или последующие биты могут быть (с большой вероятностью) равны. В связи с этим перед применением шума в криптографических приложениях необходимо, чтобы *экстракторы случайности* очистили и собрали вместе несколько его источников. Так, например, к источникам можно применить хеш-функцию и соединить полученные дайджесты операцией XOR.

Неужели это все, что относится к случайности? К сожалению, нет. Извлечение случайности из шума может быть довольно медленным процессом. Для некоторых приложений, которым требуется быстрая генерация большого количества случайных чисел, он может стать критическим ограничением. В следующем разделе будет рассказано, каким образом ОС и реальные приложения ускоряют генерацию случайных чисел.

8.2. МЕДЛЕННАЯ ГЕНЕРАЦИЯ СЛУЧАЙНОСТИ? НА ПОМОЩЬ ПРИДЕТ ГЕНЕРАТОР ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ

Случайность используется повсюду. К этому моменту у вас не должно остаться сомнений по этому поводу, по крайней мере относительно криптографии. Однако, как ни странно, криптография не единственное место, где активно задействуются случайные числа. Например, простые программы Unix, такие как ls, тоже требуют случайности! Поскольку любая ошибка в программе может иметь разрушительный эффект, двоичные файлы пытаются защититься от низкоуровневых атак с помощью множества трюков. Один из них — *ASLR* (address space layout randomization — рандомизация размещения адресного пространства) — технология, которая случайным образом меняет структуру памяти процесса при каждом запуске и которой для этого необходимы случайные числа. Другой пример — сетевой протокол TCP: при каждой установке соединения он использует случайность для получения непредсказуемой последовательности чисел и предотвращения атак, направленных на перехват соединения. Несмотря на то что все это не касается темы книги, полезно иметь представление о том, как часто в реальном мире случайность применяется для обеспечения безопасности.

В предыдущем разделе я упомянул, что получение непредсказуемой случайности, к сожалению, небыстрый процесс. Иногда это связано с тем, что источник энтропии медленно производит шум. В результате ОС часто оптимизируют производство случайных чисел, используя *генераторы псевдослучайных чисел* (ГПСЧ) (pseudorandom number generator, PRNG).

ПРИМЕЧАНИЕ

Чтобы противопоставить ГПСЧ генераторам случайных чисел, которые не предназначены для обеспечения безопасности (но полезны в различных типах приложений, например в видеоиграх), их иногда называют *криптографически стойкими* генераторами псевдослучайных чисел (cryptographically secure PRNG, CSPRNG). NIST, желая сделать все по-своему (как обычно), часто называет ГПСЧ *детерминированными генераторами случайных битов* (ДГСБ).

Для работы ГПСЧ (рис. 8.1) требуется начальный секрет, обычно называемый *зерном* (seed), который можно сформировать, смешав различные источники энтропии. Затем можно быстро произвести большое количество случайных чисел.



Рис. 8.1. ГПСЧ на основе зерна создает последовательность случайных чисел. Использование одного и того же зерна приводит к генерированию одной и той же последовательности. Не должно быть возможности восстановить состояние с помощью случайных выходных данных (один из способов — функция next). Следовательно, не должно быть возможности предсказать будущие случайные числа или восстановить ранее сгенерированные случайные числа посредством одного лишь наблюдения за созданными случайными числами

Криптостойкие ГПСЧ обычно имеют следующие свойства.

- *Детерминированность* — при использовании одного и того же зерна дважды получается одна и та же последовательность случайных чисел. Это отличается от непредсказуемого характера извлечения случайности, о котором я говорил ранее: если зерно известно, то результат генерации должен быть полностью предсказуемым. Именно поэтому данная конструкция называется псевдослучайной, и именно этим обусловлена очень высокая скорость генерации.
- *Неотличимость выхода от случайной последовательности* — на практике никто не должен отличить ГПСЧ, выдающий случайное число из множества возможных чисел, от маленькой феи, беспристрастно выбирающей случайное число из того же множества (при условии, что фея знает волшебный способ выбрать число так, чтобы оно являлось одним из всех возможных вариантов, которые могли быть выбраны с равной вероятностью). Следовательно, наблюдение за генерируемыми случайными числами само по себе не позволяет кому-либо восстановить внутреннее состояние ГПСЧ.

Последний пункт очень важен! ГПСЧ имитирует выбор *равномерно распределенного и случайного* числа, то есть каждое число из множества имеет равные шансы быть выбранным. Например, если ГПСЧ производит случайные числа

размером 8 байт, то множеством будут все возможные строки размером 8 байт и каждое восьмибайтовое значение с равной вероятностью должно быть следующим значением, которое может сформировать ГПСЧ. Сюда входят и те значения, которые были получены ГПСЧ в какой-то момент в прошлом.

Кроме того, многие ГПСЧ показывают дополнительные свойства обеспечения безопасности. К примеру, ГПСЧ способен сохранить *секретность будущих состояний* (forward secrecy): если злоумышленнику становится известно состояние ГПСЧ (допустим, в результате проникновения в компьютер жертвы), он не сможет получить уже сгенерированные случайные числа (рис. 8.2).

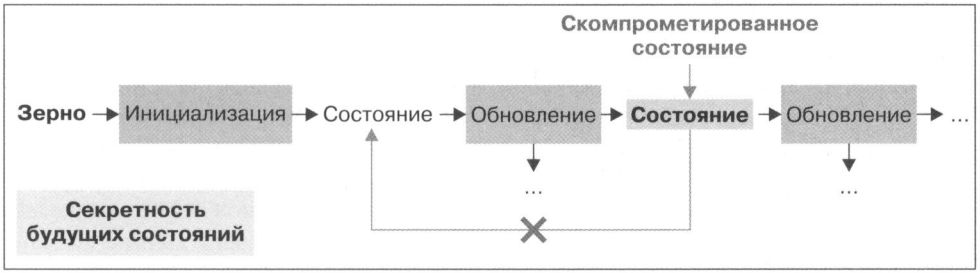


Рис. 8.2. Можно сказать, что ГПСЧ способен обеспечить секретность будущих состояний, если компрометация состояния не позволяет восстановить ранее сгенерированные случайные числа

Доступ к *состоянию* ГПСЧ означает возможность определить все псевдослучайные числа, которые станут генерироваться в будущем. Некоторые ГПСЧ устраняют эту уязвимость, применяя механизмы периодического самовосстановления (на случай компрометации). Такое восстановление может быть выполнено повторным закачиванием новой энтропии (или повторным введением зерна) после запуска ГПСЧ. Это свойство (рис. 8.3) называется *секретностью прошлых состояний* (backward secrecy).

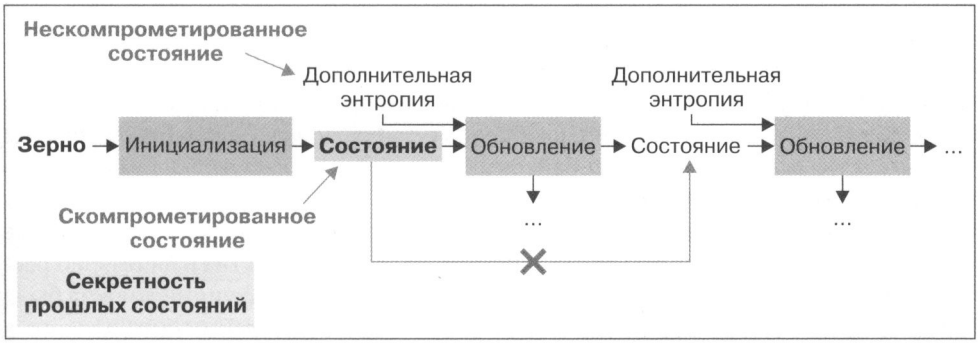


Рис. 8.3. Можно сказать, что ГПСЧ обладает свойством секретности прошлых состояний, если компрометация состояния не позволяет предсказать будущие случайные числа, генерируемые ГПСЧ. Это верно только в том случае, если после компрометации производится новая энтропия, которая вводится в функцию обновления

ПРИМЕЧАНИЕ

Термины *«секретность прошлых состояний»* и *«секретность будущих состояний»* часто вызывают путаницу. Если, читая этот раздел, вы подумали, не должна ли секретность будущих состояний быть секретностью прошлых состояний и наоборот, то вы не сошли с ума. По этой причине секретность прошлых состояний иногда называют *будущей секретностью* или даже *постскомпрометированной безопасностью* (post-compromise security, PCS).

ГПСЧ может иметь чрезвычайно высокую скорость. При наличии хорошего зерна он считается безопасным методом генерации большого количества случайных значений для криптографических целей. Очевидно, что предсказуемое или слишком маленькое число не подходит для безопасного запуска ГПСЧ. Из этого фактически следует, что у нас есть криптостойкие способы быстро растянуть большой секрет на миллиарды других закрытых ключей. Довольно круто, правда? Именно поэтому большинство криптографических приложений, если не все, не сразу используют случайные числа, извлеченные из шума, а сначала задействуют их в качестве зерна для ГПСЧ, а затем по необходимости переходят к генерированию случайных чисел из ГПСЧ.

«ЧЕРНЫЙ ХОД» DUAL EC

Сегодня ГПСЧ, как правило, представляют собой конструкции, в основе которых лежит эвристика. Это связано с тем, что конструкции, основанные на сложных математических задачах, например на дискретном логарифме, слишком медленны и поэтому непрактичны. Одним из известных примеров является *Dual EC* от АНБ, основанный на эллиптических кривых. Примерно в 2006 году ГПСЧ Dual EC был включен в различные стандарты, в том числе в некоторые публикации NIST, и вскоре после этого несколько исследователей независимо друг от друга обнаружили в нем потенциальный «черный ход». В 2013 году он был подтвержден в ходе разглашения сведений Сноуденом, а еще через год алгоритм был отозван из множества стандартов.

Зерном безопасного ГПСЧ должен быть *непредсказуемый* секрет. Если говорить конкретнее, то ГПСЧ принимает равномерно распределенный случайный ключ размером n байт. Следовательно, необходимо выбрать ключ случайным образом из множества всех возможных n-байтовых строк, где каждая строка имеет одинаковые шансы быть выбранной.

В этой книге я рассказывал о многих криптографических алгоритмах, выходные данные которых неотличимы от случайной последовательности (равномерно распределенных значений, выбранных случайным образом). Интуиция должна была вам подсказать: можно ведь использовать эти алгоритмы для генерации случайных чисел? И она права! Случайные числа можно получить с помощью хеш-функций, функции XOF, блочных и потоковых шифров и имитовставок. В теории нигде не фиксируется такое свойство выходных данных хеш-функций и MAC, как неотличимость от случайной последовательности, но на практике они часто именно

таковы. В то же время об асимметричных алгоритмах, таких как обмен ключами и подписи, почти всегда можно сказать, что они неслучайны. По этой причине их выходные данные часто хешируются перед использованием в качестве случайных чисел.

В настоящее время применение AES-CTR для формирования случайных чисел — обычное дело, поскольку AES имеет аппаратную поддержку на большинстве машин. Симметричный ключ алгоритма становится зерном, а шифротексты — случайными числами, например, для шифрования бесконечной строки из нулей. На практике для обеспечения секретности прошлых и будущих состояний эти конструкции немного усложняют. Однако теперь вы знаете достаточно для того, чтобы перейти к следующему разделу, в котором говорится о получении случайных чисел в реальных условиях.

8.3. ГЕНЕРАЦИЯ СЛУЧАЙНОСТИ НА ПРАКТИКЕ

Вы узнали о трех компонентах, необходимых ОС для предоставления криптостойких случайных чисел своим программам.

- *Источники шума* — это непредсказуемые физические явления, такие как температура устройства или движения мыши, с помощью которых ОС может получить необработанную случайность.
- *Очистка и смешивание* — так как необработанная случайность может быть плохого качества (некоторые биты искажены), ОС очищают и смешивают несколько источников и тем самым формируют хорошее случайное число.
- *ГПСЧ* — из-за небольшой скорости выполнения первых двух шагов одно равномерно распределенное случайное значение может быть использовано как зерно для ГПСЧ, который способен быстро создавать случайные числа.

В этом разделе я расскажу, как эти три концепции объединяются в упрощенные интерфейсы для разработчиков. Как правило, ОС позволяют сгенерировать случайное число, выполнив системный вызов. За таким системным вызовом стоит система, объединяющая источники шума, алгоритм смешивания и ГПСЧ (в общем виде представлена на рис. 8.4).

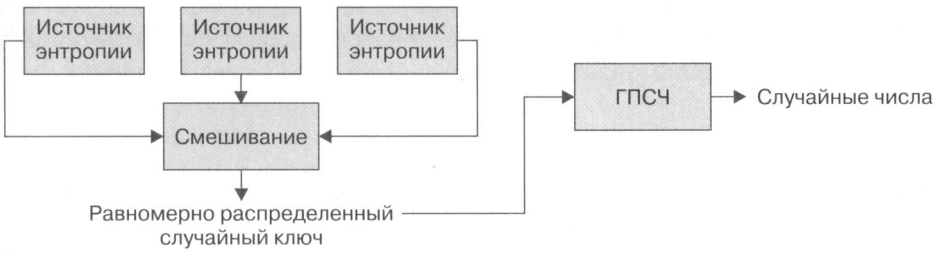


Рис. 8.4. Генерация случайных чисел в системе обычно подразумевает, что энтропия, полученная в результате смешивания различных источников шума, используется в качестве зерна для долговременного ГПСЧ

Реализация этих трех концепций зависит от ОС и имеющегося оборудования. Если говорить про 2021 год, то в Linux применяется ГПСЧ, основанный на потоковом шифре ChaCha20, а основой ГПСЧ для macOS служит хеш-функция SHA-1. Кроме того, разработчики используют тот или иной интерфейс генератора случайных чисел в зависимости от ОС, на которой работают. В Windows для получения случайных чисел можно взять системный вызов `BCryptGenRandom`. На других платформах данная функция выполняется посредством чтения файла устройства (обычно он называется `/dev/urandom`). Например, в Linux или macOS можно считать 16 байт с устройства с помощью инструмента командной строки `dd`:

```
$ dd if=/dev/urandom bs=16 count=1 2> /dev/null | xxd -p
40b1654b12320e2e0105f0b1d61e77b1
```

Одна из проблем, связанная с `/dev/urandom`, заключается в том, что обеспечиваемый им уровень энтропии может оказаться недостаточным (числа не будут достаточно случайными), если задействовать его слишком рано после загрузки устройства. В качестве решения ОС Linux и FreeBSD предложили системный вызов `getrandom`, обеспечивающий практически ту же функциональность, что и чтение `/dev/urandom`. В редких случаях, когда для инициализации ГПСЧ недостаточно энтропии, `getrandom` блокирует дальнейшую работу программы и ждет корректного зерна. По этой причине, если `getrandom` доступен в вашей системе, рекомендую использовать именно его. В листинге 8.1 показано, как можно безопасно применять `getrandom` в языке C.

Листинг 8.1. Метод получения случайных чисел в языке C

```
#include <sys/random.h>

uint8_t secret[16];
int len = getrandom(secret, sizeof(secret), 0);

if (len != sizeof(secret)) {
    abort();
}
```

Заполняет буфер произвольными байтами (обратите внимание на лимит getrandom: 256 байт за вызов)

Флаг по умолчанию (0) не допускает блокировку до правильного зерна

Если функция не сработает или вернет меньше требуемого количества случайных байтов, значит, система повреждена и лучше всего прервать выполнение

С учетом данного примера стоит также отметить, что многие языки программирования имеют стандартные и криптографические библиотеки, которые предоставляют лучшие средства. Например, легко забыть, что getrandom возвращает не более 256 байт за один вызов. По этой причине для генерации случайных чисел лучше всего подходит стандартная библиотека используемого языка программирования.

ВНИМАНИЕ

Обратите внимание на то, что во многих языках программирования существуют функции и библиотеки, позволяющие получать предсказуемые случайные числа. Они не подходят для использования в криптографических целях! Убедитесь, что вы работаете с библиотеками, которые генерируют *криптографически сильные* случайные числа. Как правило, в этом помогает само название библиотеки (например, можно догадаться, какой из пакетов Golang окажется лучшим вариантом, math/rand или crypto/rand), но ничто не заменит чтения руководства!

В листинге 8.2 показано, как сгенерировать несколько случайных байтов с помощью PHP 7. Эти случайные байты могут быть использованы в любом криптографическом алгоритме, например, в качестве секретного ключа для алгоритма аутентифицированного шифрования. Каждый язык программирования работает по-своему, поэтому генерировать случайные числа для криптографических целей лучше всего тем стандартным средством, которое указано в его документации.

Листинг 8.2. Метод получения случайных чисел на PHP

```
<?php
$bad_random_number = rand(0, 10);

$secret_key = random_bytes(16);
?>
```

Производит случайное целое число от 0 до 10. Несмотря на свою скорость, rand не генерирует криптостойких случайных чисел и поэтому не подходит для криптографических алгоритмов и протоколов

random_bytes создает и заполняет буфер 16 производными байтами. Результат подходит для использования в криптографических алгоритмах и протоколах

Теперь, когда вы узнали, как можно получить криптостойкую случайность для своих программ, давайте подумаем о том, какие требования обеспечения безопасности нужно учитывать при генерации случайности.

8.4. ГЕНЕРАЦИЯ СЛУЧАЙНОСТИ И ТРЕБОВАНИЯ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ

В этот момент следует помнить о том, что любой полезный криптографический протокол требует хорошей случайности. Взломанный ГПСЧ может привести к нарушению безопасности всего криптографического протокола или алгоритма. Несомненно, вам уже стало понятно, что имитовставка безопасна настолько, насколько безопасен используемый с ней ключ, а малейшая доля предсказуемости, как правило, разрушает такие схемы подписи, как ECDSA.

До сих пор создавалось впечатление, что генерирование случайности — простая составляющая прикладной криптографии, но на практике это не так. На самом деле применение некриптографического ГПСЧ, неправильная инициализация ГПСЧ (например, использование текущего времени, которое предсказуемо) и множество других проблем сделали из случайности источник огромного количества ошибок в реальной криптографии.

Один из примеров — программы, задействующие *ГПСЧ в пространстве пользователя*, а не *ГПСЧ в пространстве ядра*, которые лежат в основе системных вызовов. ГПСЧ в пространстве пользователя обычно добавляют ненужное трение и при неправильном применении в худшем случае могут сломать всю систему. Так, например, произошло с ГПСЧ, предлагаемым библиотекой OpenSSL: в 2006 году было выпущено исправление для некоторых ОС, которое непреднамеренно повлияло на все ключи SSL и SSH, сгенерированные с помощью уязвимого ГПСЧ.

> Удаление этого кода вызвало побочную реакцию — нарушение процесса инициализации ГПСЧ OpenSSL. Мы не добавляли произвольные данные для начального зерна, а единственным используемым случайным значением был идентификатор текущего процесса. В Linux максимальный возможный идентификатор процесса по умолчанию равен 32 768, в результате чего для всех операций ГПСЧ применяется очень небольшое количество начальных значений.
>
> *Эйч Ди Мур (Debian OpenSSL Predictable PRNG Toys, 2008)*

По этой и другим причинам, которые я упомяну далее в этой главе, разумно избегать ГПСЧ *в пространстве пользователя* и придерживаться случайности, предоставляемой ОС, если она доступна. В большинстве ситуаций достаточно того, что дает стандартная библиотека языка программирования или хорошая криптографическая библиотека.

> Мы не можем постоянно добавлять «лучшие практики» ко всему тому, что разработчики должны держать в голове при повседневном написании кода.
>
> *Мартин Босслет (OpenSSL PRNG Is Not (Really) Fork-safe, 2013)*

К сожалению, никакие советы не смогут подготовить вас к многочисленным ловушкам, связанным с формированием хорошей случайности. Поскольку случайность лежит в основе каждого криптографического алгоритма, крошечные ошибки могут привести к катастрофическим последствиям. Стоит помнить о следующих крайних случаях.

- *Процессы дублирования* — при использовании ГПСЧ в пространстве пользователя (некоторые приложения с чрезвычайно высокими требованиями к производительности могут не иметь другого варианта) важно помнить, что в результате дублирования создается новый процесс (потомок), который будет иметь то же состояние ГПСЧ, что и его родитель. Следовательно, оба ГПСЧ будут выдавать одну и ту же случайную последовательность. По этой причине, если вам действительно нужен ГПСЧ в пространстве пользователя, следует позаботиться о том, чтобы копии применяли разные зерна для своих ГПСЧ.
- *Виртуальные машины* — клонирование состояния ГПСЧ также может стать проблемой при использовании ГПСЧ ОС, а именно со стороны виртуальных машин. Если сохранить состояние виртуальной машины целиком, а затем запустить его несколько раз, то каждый экземпляр может выдать одинаковую случайную последовательность. Иногда данная проблема исправляется гипервизорами и ОС, но перед запуском приложений, запрашивающих случайные числа в виртуальных машинах, стоит проверять, что делает применяемый гипервизор.
- *Энтропия на ранней стадии загрузки* — если на пользовательских устройствах ОС не должны испытывать проблем с накоплением энтропии из шума, создаваемого взаимодействием пользователя с устройством, то на встроенных устройствах и «безголовых» системах получение хорошей энтропии во вре-

мя загрузки сопряжено с большими трудностями. Как показывает история, некоторые устройства загружаются одинаково и в итоге накапливают один и тот же начальный шум от системы, в результате чего для их внутренних ГПСЧ применяется одно и то же зерно и генерируется одна и та же случайная последовательность.

> Существует окно уязвимости — ошибка времени начальной загрузки энтропии, — в течение которого результат команды Linux urandom может быть полностью предсказуемым по крайней мере в одноядерных системах. <...> После того как мы отключили источники энтропии, которые могут быть недоступны на «безголовом» или встроенном устройстве, Linux RNG выдавал один и тот же предсказуемый поток при каждой загрузке.
>
> *Хенингер и др. (Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices, 2012)*

В редких случаях, когда очень нужно получить случайные числа на ранней стадии загрузки, можно помочь системе, обеспечив некоторую начальную энтропию, сгенерированную с помощью хорошо работающего `getrandom` или `/dev/urandom` другой машины. Такую возможность предоставляют разные ОС, и если вы окажетесь в такой ситуации, следует обратиться к их руководствам (как обычно).

Генератор истинно случайных чисел может обеспечить простое решение данной проблемы. Например, в современных процессорах Intel имеется специальный аппаратный чип, который извлекает случайность из теплового шума. Доступ к случайности предоставляет инструкция `RDRAND`.

СПОРЫ О RDRAND

Интересно, что инструкция `RDRAND` от Intel вызвала немало споров из-за страха перед риском образования «черных ходов». В большинстве ОС `RDRAND` выступает в качестве *вспомогательного* источника энтропии, так как дополнен другими источниками. Вспомогательный источник энтропии не может влиять на результат генерации случайности.

УПРАЖНЕНИЕ

Представьте на минуту, что смешивание различных источников энтропии осуществляется простой функцией XOR. Как может измениться результат?

Наконец, отмечу, что одним из способов избежать ловушек, связанных с генерацией случайности, является использование алгоритмов, работа которых *в меньшей степени* основана на случайности. Например, в главе 7 можно увидеть, что ECDSA требует генерировать случайный nonce при каждом подписании, в то время как

EdDSA — нет. Другой пример был представлен в главе 4: конструкция AES-GCM-SIV выдерживает использование одного и того же nonce дважды, в отличие от алгоритма AES-GCM, в котором произойдет утечка ключа аутентификации и последующая потеря целостности шифротекстов.

8.5. ОТКРЫТАЯ СЛУЧАЙНОСТЬ

До сих пор я по большей части говорил о *секретной случайности*, которая может понадобиться для закрытых ключей. Однако в некоторых случаях конфиденциальность не требуется и нужна *открытая случайность*. В этом разделе кратко рассмотрим некоторые способы получения такого вида случайности. Вот два сценария:

- *«один ко многим»* (one-to-many) — случайность формируется для других;
- *«многие ко многим»* (many-to-many) — несколько участников вместе формируют случайность.

Начнем с первого сценария. Представьте, что вам нужно сгенерировать случайный поток таким образом, чтобы многие участники могли его проверить. Другими словами, поток должен быть непредсказуемым и неизменяемым с вашей стороны. Далее представьте, что у вас есть схема подписи, которая предоставляет уникальные подписи, созданные на основе пары ключей и сообщения. Схему дополняет конструкция, которая позволяет получать случайные числа, поддающиеся проверке, — функция генерации проверяемых случайных чисел (a verifiable random function, VRF). Вот как она работает (рис. 8.5).

1. Генерируется пара ключей. В открытом доступе публикуются ключ проверки и зерно.
2. Для генерации случайных чисел подписывается открытое зерно, а получившаяся подпись хешируется и публикуется в открытом доступе. Дайджест является тем самым случайным числом, а подпись выступает в качестве доказательства.
3. Любой может захешировать подпись и посмотреть, совпадает ли результат со случайным числом. Для проверки правильности подписи ему понадобятся открытое зерно и ключ проверки.

Эта конструкция может быть расширена для формирования множества случайных чисел путем использования открытого зерна как счетчика. Поскольку подпись уникальна, а открытое зерно фиксировано, у подписывающего нет возможности сгенерировать другое случайное число.

УПРАЖНЕНИЕ

Схемы подписи типа BLS, упомянутые на рис. 8.5 и в главе 7, создают уникальные подписи, но они не подходят для ECDSA и EdDSA. Почему?

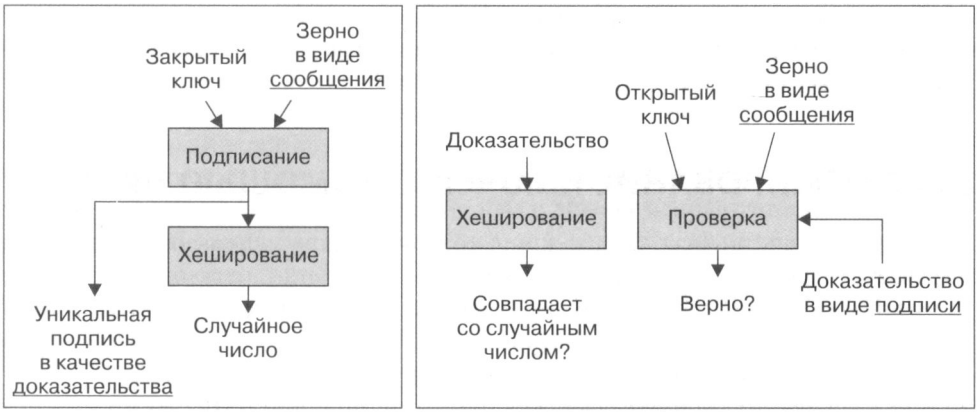


Рис. 8.5. Функция генерации проверяемых случайных чисел использует криптографию с открытым ключом для генерации проверяемой случайности, которую можно проверить. Сгенерировать случайное число можно следующим образом: просто подписать зерно с помощью какой-нибудь схемы подписи, производящей уникальные подписи (например, BLS), а затем захешировать подпись. В результате получится открытое случайное число. Для подтверждения полученной случайности необходимо удостовериться в том, что хеш подписи действительно является тем самым случайным числом, а также проверить подпись, добавленную к зерну

Для решения этой проблемы в Internet Draft (документ, который должен стать RFC) — https://tools.ietf.org/html/draftirtfcfrgvrf08 — указано, как реализовать VRF с помощью ECDSA. В некоторых сценариях, например в лотерейной игре, несколько участников могут захотеть случайным образом определить победителя. Мы говорим о *децентрализованных маяках случайности*. Их роль состоит в том, чтобы производить одинаковую проверяемую случайность, даже если некоторые участники решат не принимать участия в протоколе. Общим решением является использование рассмотренного ранее VRF, но не с одним ключом, а с *пороговым распределенным ключом*. Такой ключ делится между многими участниками и выдает уникальную действительную подпись для определенного сообщения только после того, как его подпишет пороговое количество участников. Все это звучит немного сложно, поскольку я впервые говорю о распределенных ключах. Однако вскоре вы узнаете о них больше.

Одним из популярных децентрализованных маяков случайности является *drand*. Над ним совместно работают несколько организаций и университетов. Он доступен по адресу https://leagueofentropy.com.

> Основная проблема, возникающая при создании хорошей случайности, заключается в том, что ни одна из сторон, участвующих в процессе генерации случайности, не должна иметь возможности предсказать конечный результат или повлиять на него. Сеть Drand не контролируется ни одним из ее участников. В ней нет единой точки отказа, и никто из операторов серверов drand не может повлиять на случайность, сгенерированную сетью.
>
> *https://drand.love (How drand works, 2021)*

Теперь, когда я подробно рассказал о случайности и современных способах ее получения, перейдем к роли секретов в криптографии и к методам управления ими.

8.6. ФОРМИРОВАНИЕ КЛЮЧЕЙ С ПОМОЩЬЮ HKDF

ГПСЧ — не единственная конструкция, которую можно использовать для формирования нескольких секретов на основе одного секрета (другими словами, для растягивания ключа). Формирование нескольких секретов из одного — настолько частый паттерн в криптографии, что у этого понятия есть собственное название — *формирование ключа* (key derivation). Итак, посмотрим, о чем идет речь.

Функция формирования ключа (key derivation function, KDF) во многом похожа на ГПСЧ, за исключением ряда тонкостей, указанных далее. Эти различия наглядно обобщены на рис. 8.6.

- Равномерно распределенный случайный секрет (при условии, что он обладает достаточной энтропией) — не обязательное условие для KDF. Это делает KDF полезным для формирования секретов из выходных значений обмена ключами, который дает высокую энтропию, но искаженные результаты (см. главу 5). Полученные секреты, в свою очередь, являются равномерно распределенными случайностями, поэтому их можно задействовать в конструкциях, которым необходимы равномерно распределенные случайные ключи.
- KDF обычно используется в протоколах, которые требуют от участников неоднократного формирования одних и тех же ключей. В этом смысле KDF должен обладать детерминированностью, в то время как ГПСЧ иногда обеспечивают секретность прошлых состояний за счет частого накачивания себя энтропией.
- Как правило, KDF не предназначен для получения большого количества случайных чисел. Обычно он применяется для получения ограниченного количества ключей.

Наиболее популярный вид KDF — это *функция формирования ключа на основе HMAC (HKDF)*. О HMAC (MAC, основанном на хеш-функциях) рассказывалось в главе 3. HKDF определена в стандарте RFC 5869 и представляет собой упрощенную версию KDF внутри HMAC. По этой причине HKDF можно использовать с различными хеш-функциями, хотя чаще всего она применяется с SHA-2. HKDF представляет собой две функции:

- *HKDF-Extract* — удаляет искажения из секретных входных данных, производя равномерно распределенный случайный секрет;
- *HKDF-Expand* — производит равномерно распределенные случайные выходные данные произвольной длины. Как и ГПСЧ, *в качестве входных данных требует равномерно распределенный случайный секрет* и поэтому обычно запускается после выполнения HKDF-Extract.

Сначала рассмотрим функцию HKDF-Extract, изображенную на рис. 8.7. Технически, чтобы унифицировать случайность входной строки байтов, достаточно какой-нибудь хеш-функции (помните, что выход хеш-функции должен быть неотличим от случайного), но HKDF идет дальше и принимает еще одно входное значение — *соль*. Как и в случае с хешированием паролей, соль позволяет дифференцировать разные варианты применения HKDF-Extract в одном и том же протоколе. Несмотря на то что она необязательна и, если не используется, представляет собой строку, состоящую только из нулевых байтов, все же не рекомендуется ее игнорировать. Более того, HKDF не предполагает секретности соли, она может быть известна всем без исключения. Вместо хеш-функции HKDF-Extract задействует имитовставку (в частности, HMAC), интерфейс которой по совпадению принимает два аргумента.

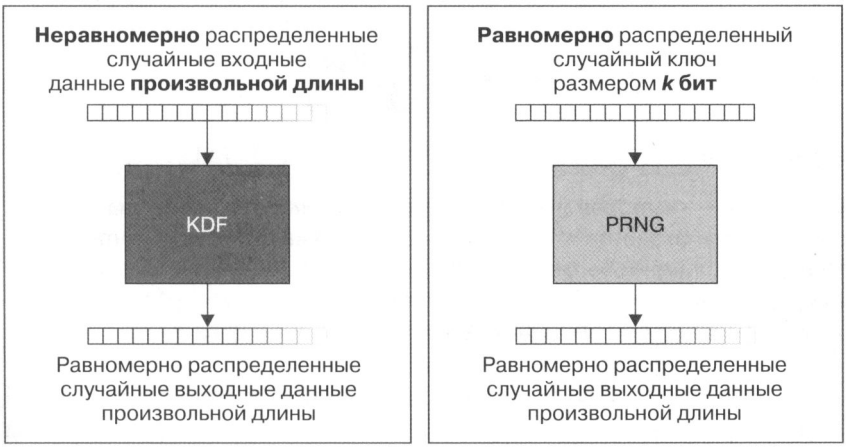


Рис. 8.6. Функция формирования ключа и ГПСЧ — похожие конструкции. Основные различия между ними заключаются в том, что KDF не требует полностью равномерно распределенного случайного секрета (при условии, что он обладает достаточной энтропией) в качестве входного значения и обычно не используется для генерации слишком большого количества выходных данных

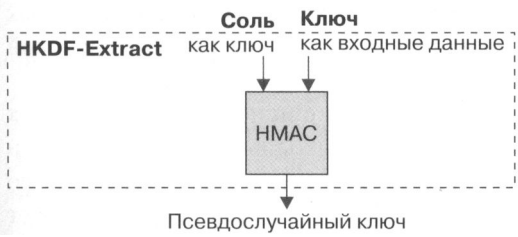


Рис. 8.7. HKDF-Extract — это первая функциональная часть HKDF. На входе она принимает необязательную соль в качестве ключа для HMAC и секрет, который может быть неравномерно распределенной случайностью. Использование разных солей с одним и тем же секретом дает разные результаты

Теперь рассмотрим HKDF-Expand (рис. 8.8). Если секрет на входе уже является равномерно распределенной случайностью, HKDF-Extract можно пропустить и сразу переходить к HKDF-Expand.

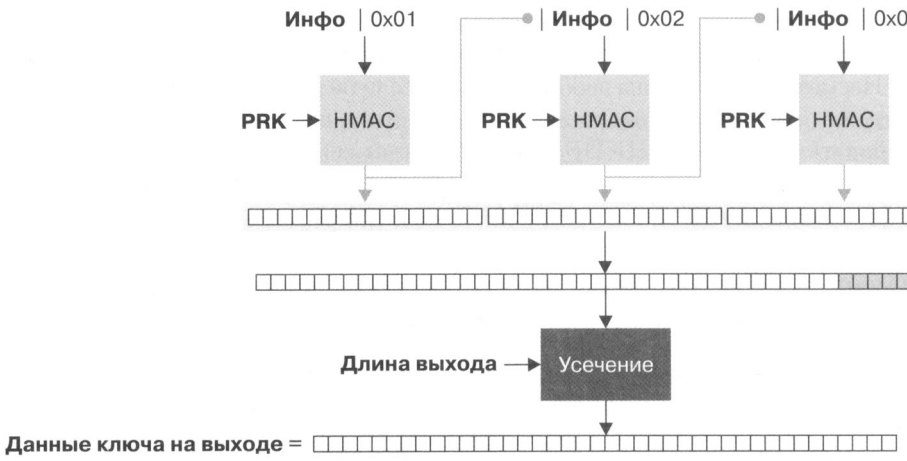


Рис. 8.8. HKDF-Expand — это вторая функциональная часть HKDF. На входе она принимает необязательную байтовую строку информации в качестве аргумента info и секрет, который должен быть равномерно распределенной случайностью. Использование разных аргументов info с одним и тем же секретом дает разные результаты. Длина выходных данных контролируется аргументом length

Подобно HKDF-Extract, HKDF-Expand тоже принимает дополнительный и необязательный аргумент настройки `info`. В то время как соль предназначена для обеспечения некоторого разделения доменов между вызовами в рамках одного протокола HKDF (или HKDF-Extract), `info` используется для дифференцирования определенной версии HKDF (или HKDF-Expand) от других протоколов. К тому же можно указать, какой объем выходных данных необходимо получить. Тем не менее не стоит забывать, что HKDF не является ГПСЧ и не предназначен для получения большого количества секретов. Алгоритм ограничен размером используемой хеш-функции, то есть если HKDF применяется вместе с SHA-512 (размер его выходных данных — 512 бит), то размер выходных данных для заданного ключа и байтовой строки `info` не превысит 512 × 255 бит = 16 320 байт.

Вызов HKDF или HKDF-Expand несколько раз с использованием одних и тех же аргументов, но с указанием разной длины выходных данных дает один и тот же результат, усеченный до запрашиваемой длины (рис. 8.9). Это свойство называется *соответствующими выходными данными* и в редких случаях может удивить разработчиков протоколов, поэтому о нем полезно знать.

Большинство криптографических библиотек объединяют HKDF-Extract и HKDF-Expand в один вызов (рис. 8.10). Как обычно, перед использованием HKDF обязательно следует прочитать руководство — в данном случае RFC 5869.

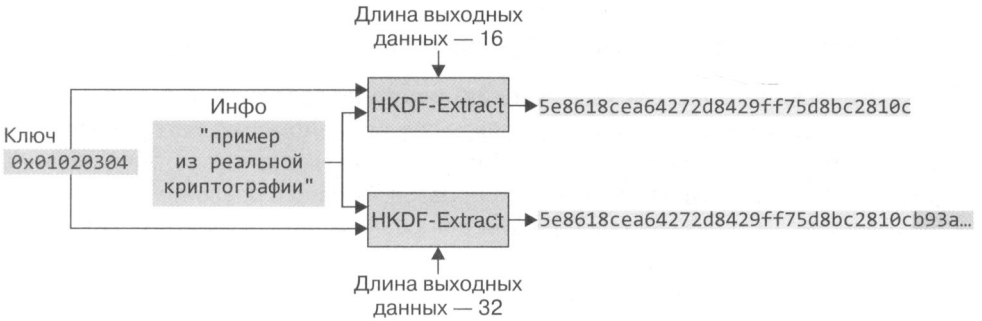


Рис. 8.9. HKDF и HKDF-Expands предоставляют соответствующие выходные данные, то есть размер выходных данных усекается в соответствии со значением длины, заданным при вызове функции

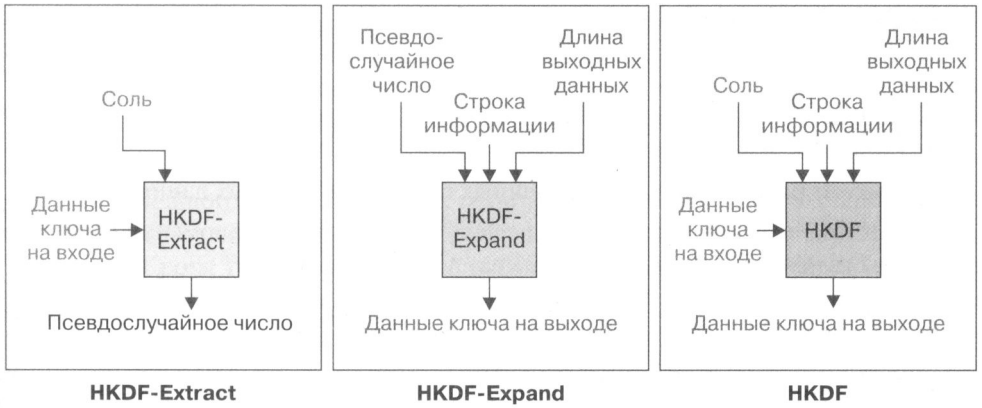


Рис. 8.10. HKDF обычно реализуется в виде одного вызова функции, сочетающего в себе HKDF-Extract (для извлечения равномерно распределенной случайности из входного ключа) и HKDF-Expand (для генерации выходных данных произвольной длины)

HKDF — не единственный способ получить несколько секретов из одного. Более примитивный подход заключается в использовании *хеш-функций*. Поскольку они не требуют равномерно распределенной случайности на входе и выдают равномерно распределенную случайность на выходе, то вполне годятся для этой задачи. Однако применение хеш-функций — далекое от идеала решение, поскольку их интерфейс не учитывает разделение доменов (нет аргумента строки настройки), а длина выходных данных фиксирована. Таким образом, если есть выбор, стоит отдать предпочтение KDF. Тем не менее некоторые общепризнанные алгоритмы используют именно хеш-функции. Например, схема подписи Ed25519 из главы 7 хеширует 256-битный ключ с помощью SHA-512 для получения двух 256-битных ключей.

ДЕЙСТВИТЕЛЬНО ЛИ ЭТИ ФУНКЦИИ ВЫДАЮТ СЛУЧАЙНЫЕ РЕЗУЛЬТАТЫ?

Теоретически в свойствах хеш-функции ничего не говорится о том, что ее выходные данные должны представлять собой равномерно распределенную случайную последовательность. Они лишь требуют от нее устойчивости к коллизиям и атакам нахождения прообразов. Однако в реальном мире мы повсеместно используем хеш-функции для реализации случайного оракула (см. главу 2), и по этой причине предполагается, что их выходные данные — это равномерно распределенная случайная последовательность. То же самое верно и для имитовставок: теоретически они не должны выдавать на выходе равномерно распределенные случайные данные (в отличие от PRF из главы 3), но на практике чаще всего происходит именно это. Вот почему HKDF использует HMAC. Таким образом, в остальной части книги будет подразумеваться, что популярные хеш-функции (например, SHA-2 и SHA-3) и MAC (например, HMAC и KMAC) выдают случайные результаты.

Функции расширенного выхода (XOF), которые мы рассматривали в главе 2 (SHAKE и cSHAKE), можно применять и в качестве KDF! Вспомните, что XOF:

- не требует равномерно распределенных случайных входных данных;
- может производить практически бесконечно длинные, равномерно распределенные случайные выходные данные.

Кроме того, KMAC (MAC из главы 3) не имеет проблемы с соответствующими выходными данными, о которой упоминалось ранее. Аргумент длины KMAC рандомизирует результат алгоритма, эффективно действуя как дополнительная строка настройки.

Наконец, существует крайний случай для входных данных с низкой энтропией. В качестве примера можно представить пароли, которые относительно легко угадываются по сравнению со 128-битным ключом. Функции формирования ключей на основе паролей в качестве метода хеширования паролей (см. главу 2) также можно использовать для получения ключей.

8.7. УПРАВЛЕНИЕ КЛЮЧАМИ И СЕКРЕТАМИ

Итак, мы знаем, как генерировать криптографические случайные числа и формировать секреты в различных ситуациях. Но это еще не все.

Теперь, когда в нашем распоряжении имеются все эти криптографические алгоритмы, перед нами возникают вопросы, связанные с хранением множества секретных ключей. Как их хранить? Как защитить такие чрезвычайно важные секреты от компрометации? И что делать, если секрет все же будет скомпрометирован? Совокупность этих вопросов представляет собой *управление ключами*.

> Криптография — это инструмент, позволяющий связать целый комплекс задач с управлением ключами.
>
> *Лия Кисснер (http://mng.bz/eMrJ, 2019)*

Многие системы предпочитают оставлять ключи рядом с приложениями, которые их используют. Однако это совсем не значит, что у приложений нет никаких средств защиты на случай возникновения проблем. Для подготовки к возможному взлому или появлению ошибки, которая приведет к утечке ключа, большинство серьезных приложений реализуют две техники защиты:

- *ротацию ключей* — добавление срока действия к ключу (как правило, открытому) и периодическая замена ключа на новый позволяет «излечиться» от возможной компрометации. Чем короче срок действия и чаще ротации, тем быстрее будет заменен ключ, который стал известен злоумышленнику;
- *отмену ключей* — ротация ключей не всегда оказывается достаточной мерой защиты, и может возникнуть необходимость прекращения действия ключа при обнаружении его компрометации. По этой причине некоторые системы позволяют пользователям узнать об отмене ключа перед его применением. (Подробнее об этом вы узнаете в следующей главе, посвященной безопасной транспортировке.)

Для успешного применения этих техник зачастую просто необходима автоматизация, ведь отлаженная машина гораздо чаще работает правильно в критические моменты. Кроме того, можно присвоить ключу определенную роль и тем самым ограничить последствия компрометации. Например, в каком-нибудь вымышленном приложении можно разграничить два открытых ключа: открытый ключ 1, предназначенный только для подписания транзакций, и открытый ключ 2, предназначенный только для обмена ключами. Таким образом, компрометация закрытого ключа, связанного с открытым ключом 2, не повлияет на подписание транзакций.

Если вы не хотите оставлять ключи на каких-либо съемных носителях, то воспользуйтесь аппаратными решениями, которые предотвращают извлечение ключей. Подробнее о них вы узнаете в главе 13, посвященной аппаратной криптографии.

Наконец, существует множество способов делегировать приложениям управление ключами. В мобильных ОС часто используются *хранилища ключей*, или *связки ключей*, которые не только хранят ключи, но и выполняют шифрование!

В отдельных случаях облачные приложения могут иметь доступ к облачным сервисам управления ключами. Приложения перекладывают на такие сервисы обязанность по созданию секретных ключей, шифрованию и защите от многочисленных атак. Тем не менее, как и в случае с аппаратными вариантами, если приложение было скомпрометировано, оно все равно сможет выполнить любой запрос к делегированному сервису.

ПРИМЕЧАНИЕ

Не существует единственно верных решений, поэтому необходимо как следует продумать возможные действия для обнаружения компрометации и реагирования на нее.

Управление ключами — сложная задача, которая выходит за рамки данной книги, поэтому не буду слишком подробно останавливаться на этой теме. В следующем разделе я расскажу о криптографических техниках, позволяющих избежать проблемы управления ключами.

8.8. ДЕЦЕНТРАЛИЗАЦИЯ ДОВЕРИЯ С ПОМОЩЬЮ ПОРОГОВОЙ КРИПТОГРАФИИ

Управление ключами — это обширная область исследований, инвестируют в которую неохотно, поскольку пользователям часто не хватает ни ресурсов для внедрения лучших практик, ни доступных инструментов. К счастью, криптографии есть что предложить тем, кто хочет облегчить для себя бремя управления ключами. Первое, о чем я расскажу, — это *разделение секрета*. Оно позволяет распределить секрет между несколькими участниками, разбив его на несколько частей. Секретом здесь может быть что угодно: симметричный ключ, закрытый ключ подписи и т. д.

Обычно человек, *дилер*, генерирует секрет, разделяет его и получившиеся доли распределяет между всеми участниками, после чего удаляет секрет. Самая известная схема разделения секрета была придумана Ади Шамиром (один из изобретателей RSA) и в дальнейшем названа в его честь (Shamir's Secret Sharing, SSS) (рис. 8.11).

Когда секрет нужен для выполнения какой-либо криптографической операции (шифрования, подписи и т. д.), все участники должны вернуть свои доли дилеру, который отвечает также за восстановление исходного секрета. Такая схема делает бесполезной атаку на отдельного пользователя, поскольку каждая доля сама по себе ничего не значит. Для того чтобы получить возможность воспользоваться ключом, злоумышленники вынуждены будут скомпрометировать всех участников! Эта схема наглядно изображена на рис. 8.12.

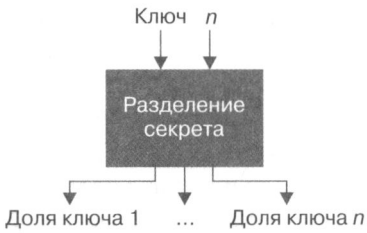


Рис. 8.11. Схема разделения секрета Шамира создает n-е количество долей ключа того же размера, что и исходный ключ



Рис. 8.12. Для восстановления исходного ключа схема разделения секрета Шамира требует наличия всех n его долей

Математика, лежащая в основе алгоритма данной схемы, на самом деле не слишком сложна! Поэтому позвольте мне в нескольких абзацах дать упрощенное представление об этой схеме.

Представьте случайную прямую линию в двумерном пространстве. Предположим, что ее уравнение $y = ax + b$ является секретом. Если два участника знают две случайные точки на этой линии, они вместе смогут восстановить ее уравнение (рис. 8.13). Эта схема работает для многочленов любой степени и, таким образом, может быть использована для разделения секрета на произвольное количество долей.

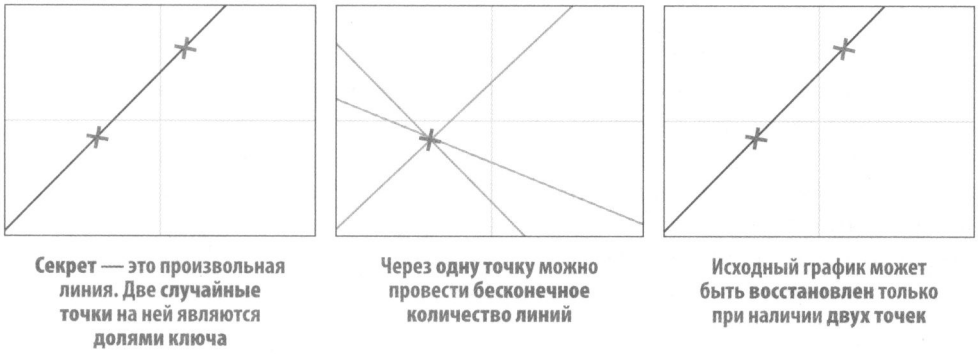


Рис. 8.13. Идея схемы разделения секрета Шамира заключается в том, чтобы рассматривать многочлен, задающий график, в качестве секрета, а случайные точки на графике — как доли ключа. Для восстановления многочлена n-й степени, задающего график, нужно знать n + 1 точку на нем. Например, $f(x) = 3x + 5$ имеет степень 1, поэтому для восстановления многочлена нужны любые две точки $(x, f(x))$, а $f(x) = 5x^2 + 2x + 3$ имеет степень 2, следовательно, нужны любые три точки

Разделение секрета — простая и поэтому популярная техника. Однако при каждом использовании в какой-нибудь криптографической операции этот ключ должен быть восстановлен, то есть его доли должны быть собраны вместе. Так появляется окно возможностей, в котором секрет становится уязвимым для кражи или случайной утечки, что фактически возвращает нас к модели *единой точки отказа*. Обойти проблему единой точки отказа можно с помощью нескольких криптографических техник, которые могут быть полезны в различных сценариях.

Например, представьте протокол, который принимает финансовую транзакцию только в том случае, если она подписана Алисой. Это возлагает на нее тяжелое бремя, ведь она может стать мишенью для злоумышленников. Для того чтобы уменьшить влияние атаки на Алису, можно изменить протокол так, чтобы он принимал (для одной и той же транзакции) n-е количество подписей от n-го количества

различных открытых ключей, включая ключ Алисы. Для подделки действительной транзакции злоумышленнику придется скомпрометировать все n подписей! Такие системы называются *мультиподписями* и широко распространены в криптовалютном пространстве.

Впрочем, примитивные схемы мультиподписи могут привести к нежелательным затратам ресурсов. Фактически размер транзакции в нашем примере растет вместе с количеством необходимых подписей. Некоторые схемы подписи, например BLS, решают данную проблему сжатием нескольких подписей в одну. Такое действие называется *объединением подписей*. Другие схемы идут еще дальше и позволяют объединить n-е количество открытых ключей в единый открытый ключ. Эта техника называется *генерацией распределенного ключа* (distributed key generation, DKG) и является частью области криптографии под названием *«протокол конфиденциального вычисления»* (secure multi-party computation), о котором я расскажу в главе 15.

Генерация распределенного ключа позволяет n-му количеству участников совместно вычислить открытый ключ без соответствующего ему закрытого ключа (в отличие от схемы подписи Шамира, здесь нет дилера). Если нужно подписать сообщение, участники могут совместно создать подпись, используя свои секретные доли. Проверка этой подписи может быть выполнена с помощью созданного ранее открытого ключа. Опять же закрытого ключа физически не существует, что предотвращает проблему единой точки отказа, присущую схеме подписи Шамира. Поскольку в главе 7 вы познакомились с подписями Шнорра, то поймете наглядное представление упрощенной схемы DKG на основе подписей Шнорра (рис. 8.14).

Обратите внимание на следующие особенности приведенных схем.

- Каждая из схем может работать даже тогда, когда в протоколе принимает участие только пороговое число m из n-го количества участников. Это очень важное замечание, поскольку большинство реальных систем должны допускать присутствие некоторого количества злоумышленников или неактивных участников.
- Схемы этого типа могут работать с другими асимметричными криптографическими алгоритмами. Например, посредством порогового шифрования группа участников может асимметрично расшифровать сообщение.

Все эти примеры представлены на рис. 8.15.

Пороговые схемы — важная новая парадигма в мире управления ключами, и следить за их развитием полезно. В настоящее время в NIST существует группа, занимающаяся развитием пороговой криптографии. Целью группы в долгосрочной перспективе, помимо организации семинаров, является стандартизация соответствующих примитивов и протоколов.



Рис. 8.14. Схема подписи Шнорра может быть децентрализована в схему генерации распределенного ключа

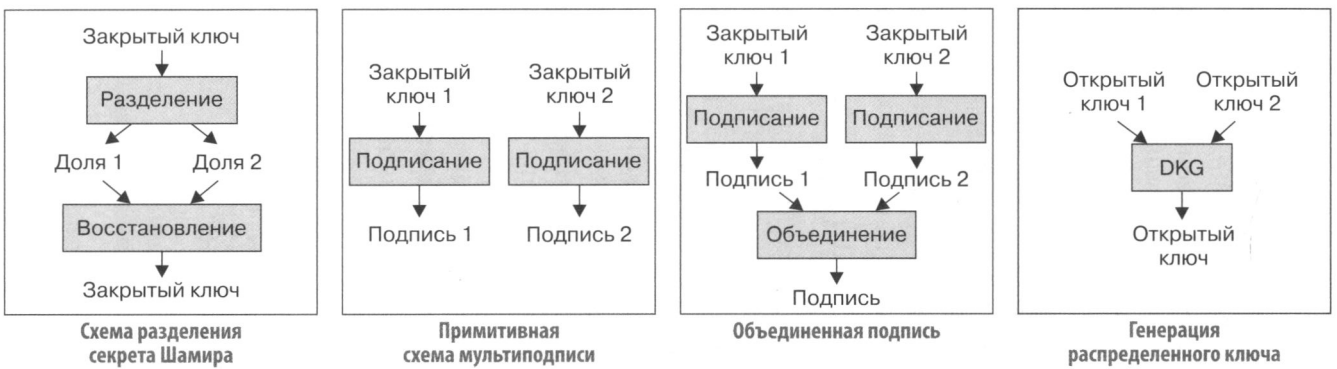


Рис. 8.15. Обзор существующих методов разделения доверия на несколько участников

РЕЗЮМЕ

- Число считается равномерно распределенным и случайным, если вероятность его выбора равна вероятности выбора всех других чисел множества.
- Энтропией называют метрику, показывающую, насколько случайной является байтовая строка. Высокая энтропия означает, что байтовая строка является равномерно распределенной случайностью; под низкой энтропией подразумевают, что байтовую строку легко угадать или предсказать.
- Генераторы псевдослучайных чисел — это алгоритмы, которые используют равномерно распределенное случайное значение в качестве зерна и при условии достаточно большого размера последнего генерируют (на практике) почти бесконечное количество случайностей, которые можно применять в криптографических целях, например, в качестве криптографических ключей.
- При формировании случайных чисел следует использовать средства, предлагаемые стандартной библиотекой применяемого языка программирования или известными криптографическими библиотеками. Если они недоступны, можно обратиться к самим операционным системам, которые обычно предоставляют интерфейсы генерации случайных чисел:
  - Window предлагает системный вызов `BCryptGenRandom`;
  - Linux и FreeBSD предлагают системный вызов `getrandom`;
  - в других Unix-подобных операционных системах обычно есть специальный файл `/dev/urandom` для генерации случайности.
- Функции формирования ключа (KDF) полезны в тех случаях, когда необходимо сформировать секреты из искаженного, но обладающего высокой энтропией секрета.
- HKDF (функция формирования ключа на основе HMAC) — это наиболее распространенный вариант KDF.
- Управление ключами подразумевает сохранение секретов в тайне и состоит из поиска места хранения секретов, проактивной установки срока действия, ротации секретов, определения дальнейших действий после компрометации секретов и т. д.
- Для уменьшения бремени управления ключами можно перераспределить доверие с одного участника на нескольких.

Часть II

Протоколы: рецепты применения криптографии

В этой части книги рассмотрим примеры использования того, о чем говорилось в части I. Если криптографические примитивы, с которыми вы познакомились ранее, являются основными ингредиентами криптографии, то теперь вам предстоит изучить рецепты их применения. А готовить придется очень много! Если во времена Цезаря криптография использовалась для шифрования сообщений, то сегодня она повсюду и уследить за всеми способами ее применения довольно сложно.

В главах 9–11 я расскажу о том, где и как криптография задействуется для решения реальных проблем, в частности, как происходит шифрование сообщений и аутентификация участников протоколов. По большому счету, использование криптографии сводится именно к этому. Участников может быть много или мало, они могут состоять из битов или из плоти. Как вы скоро убедитесь, в реальном мире применение криптографии сопряжено с компромиссами и в зависимости от контекста решения могут различаться.

Главы 12 и 13 посвящены таким быстро развивающимся областям криптографии, как криптовалюты и аппаратная криптография. Первую тему авторы большинства книг по криптографии игнорируют. (Я считаю, что данная книга является первой из посвященных криптографии, в которой есть глава о криптовалютах.) Тема аппаратной криптографии также часто упускается из виду: криптографы обычно предполагают, что их примитивы и протоколы работают в доверенной среде, что становится все более редким явлением. Аппаратная криптография расширяет границы возможного применения криптографии и позволяет гарантировать безопасность по мере того, как злоумышленники подбираются к вам все ближе и ближе.

В главах 14 и 15 речь идет о передовых технологиях, которые уже есть или вот-вот появятся. Из них вы узнаете о постквантовой криптографии — области криптографии, которая может оказаться полезной в случае изобретения масштабируемых квантовых компьютеров. Эти компьютеры, основанные на новых парадигмах, пришедших из области квантовой физики, могут произвести революцию в сфере исследований и, возможно, даже разрушить наш криптомир... Вы также узнаете о том, что я называю криптографией нового поколения, а именно о криптографических примитивах, которые редко использовались ранее, но которые вы, скорее всего, будете видеть все чаще по мере их изучения, повышения эффективности и внедрения разработчиками приложений. И наконец, в главе 16 я приведу заключительные замечания по поводу применения криптографии в реальном мире и некоторые соображения этического характера.

9 Безопасная передача данных

В этой главе

- ✓ Безопасные транспортные протоколы.
- ✓ Протокол защиты транспортного уровня (TLS).
- ✓ Протокольный фреймворк Noise.

В настоящее время криптография наиболее активно используется для шифрования сообщений, поскольку изначально она была изобретена именно для этого. Однако приложения, как правило, не задействуют такие криптографические примитивы, как аутентифицированное шифрование, напрямую, а вместо этого применяют гораздо более сложные протоколы, которые абстрагируют применение криптографических примитивов. За неимением лучшего термина я называю их *безопасными транспортными протоколами*.

В этой главе вы узнаете о наиболее распространенном из них, а именно о протоколе защиты транспортного уровня (Transport Layer Security, TLS). Мы кратко рассмотрим и другие протоколы безопасной передачи данных и их отличия от TLS.

9.1. ПРОТОКОЛЫ SSL И TLS

Чтобы понять важность *транспортных протоколов* (используемых для шифрования сообщений, которыми обмениваются машины), рассмотрим конкретный сценарий. Когда вы заходите на сайт вроде http://example.com, ваш браузер применяет несколько протоколов для установления соединения с веб-сервером и получения запрошенной страницы. Один из них — *протокол передачи гипертекста* (Hypertext

Transfer Protocol, HTTP), с помощью которого браузер сообщает веб-серверу о том, какая страница его интересует. Протокол HTTP использует человекочитаемый формат. Это означает, что вы можете просматривать и читать передаваемые HTTP-сообщения без помощи сторонних инструментов. Однако этого недостаточно для обеспечения взаимодействия между вашим браузером и веб-сервером.

HTTP-сообщения инкапсулируются в сообщения другого типа, называемые *TCP-кадрами*, которые определены в протоколе управления передачей (Transmission Control Protocol, TCP). Протокол TCP использует двоичный формат, который не является человекочитаемым. Чтобы прочитать содержимое полей TCP-кадра, требуется специальный инструмент. В свою очередь, TCP-сообщения инкапсулируются с помощью протокола межсетевого взаимодействия (Internet Protocol, IP), а IP-сообщения — с помощью чего-то еще. Все вместе это называется *стеком интернет-протоколов*, и поскольку этой теме посвящено множество книг, я не буду в нее углубляться.

Вернемся к нашему сценарию и обсудим вопрос конфиденциальности. Любой, кто сидит на проводе между вашим браузером и веб-сервером example.com, может пассивно наблюдать и читать ваши запросы и ответы сервера. Хуже того, злоумышленники, реализующие атаки посредника, могут активно подделывать и переупорядочивать эти сообщения.

Представьте, что данные вашей кредитной карты утекают каждый раз, когда вы покупаете что-то в Интернете, пароли оказываются украденными, когда вы заходите на сайт, ваши фотографии и личные сообщения попадают в руки третьих лиц, когда вы отправляете их друзьям, и т. д. В ответ на опасения множества людей по этому поводу в 1990-х годах появился предшественник TLS — *протокол защищенных сокетов* (Secure Sockets Layer, SSL). Хотя протокол SSL может применяться в различных ситуациях, изначально он предназначался для веб-браузеров. Результатом его использования в сочетании с HTTP стало расширение *Hypertext Transfer Protocol Secure* (HTTPS), позволявшее браузерам защищать свои соединения с посещаемыми веб-сайтами.

9.1.1. От SSL к TLS

Хотя SSL был не единственным протоколом, направленным на защиту части веб-ресурсов, он привлек к себе наибольшее внимание и со временем стал фактическим стандартом. Но на этом история не заканчивается. Между выходом первой и современной версий SSL произошло множество событий. Все версии SSL (последней является SSL v3.0) были взломаны по причине неудачного дизайна и использования плохих криптографических алгоритмов. (Многие из атак были описаны в документе RFC 7457.)

После выхода SSL 3.0 протокол был официально передан Инженерному совету Интернета (Internet Engineering Task Force, IETF) — организации, отвечающей за публикацию документов RFC (Request For Comments — запрос на комментарии). Название SSL было изменено на TLS, версия TLS 1.0 была выпущена в 1999 году и описана в документе RFC 2246. Последней версией TLS является TLS 1.3, описанная в документе RFC 8446 и опубликованная в 2018-м. В отличие от своей

предшественницы версия TLS 1.3 — это результат тесного сотрудничества между представителями индустрии и научными кругами. Тем не менее сейчас в Интернете все еще используется множество различных версий SSL и TLS, поскольку серверы не спешат обновляться.

ПРИМЕЧАНИЕ

Названия SSL и TLS создают большую путаницу. Сейчас протокол называется TLS, но во многих статьях и даже библиотеках по-прежнему в ходу название SSL.

TLS — не просто протокол, обеспечивающий безопасность во Всемирной паутине, в настоящее время он используется в различных сценариях и многими типами приложений и устройств в качестве протокола для защиты коммуникаций. Таким образом, то, что вы узнаете о TLS в этой главе, пригодится вам не только в Интернете, но и в любом сценарии, где требуется обеспечить безопасную связь между двумя приложениями.

9.1.2. Практическое применение протокола TLS

Прежде чем переходить к способу использования протокола TLS, определим основные термины. Два участника, которые хотят защитить свои коммуникации, называются *клиентом* и *сервером*. Это работает так же, как и в других сетевых протоколах наподобие TCP или IP: клиент инициирует соединение, а сервер ожидает, когда оно будет инициировано. Как правило, TLS-клиент состоит:

- из *некоторой конфигурации*. Клиент конфигурируется с указанием версий SSL и TLS, которые он собирается поддерживать, криптографических алгоритмов, которые будет использовать для защиты соединения, способов аутентификации серверов и т. д.;
- *некоторой информации о сервере, к которому он хочет подключиться*. Как минимум она включает IP-адрес и порт, а в случае с Интернетом — полное доменное имя (например, example.com).

При наличии этих двух аргументов клиент может инициировать соединение с сервером для организации защищенного *сеанса* — канала связи, по которому клиент и сервер могут обмениваться зашифрованными сообщениями. В некоторых случаях установка безопасного соединения обрывается на полпути. Например, если в процесс пытается вмешаться злоумышленник или конфигурация сервера оказывается несовместимой с конфигурацией клиента (подробнее об этом — чуть позже), последнему не удается организовать защищенный сеанс.

С TLS-сервером, как правило, все намного проще, поскольку ему требуется создать конфигурацию, аналогичную конфигурации клиента. Затем сервер ждет, пока клиенты подключатся к нему для организации защищенного сеанса. В практическом отношении использование протокола TLS на стороне клиента может быть настолько же простым, как показано в листинге 9.1 (конечно, если вы работаете с таким языком программирования, как Golang).

Листинг 9.1. TLS-клиент на языке Golang

```
import "crypto/tls"

func main() {
    destination := "google.com:443"
      TLSconfig := &tls.Config{}
      conn, err := tls.Dial("tcp", destination, TLSconfig)
    if err != nil {
        panic("failed to connect: " + err.Error())
    }
    conn.Close()
}
```

Полное доменное имя и порт сервера (порт 443 используется в HTTPS по умолчанию)

Пустая структура Config служит конфигурацией по умолчанию

Как клиент узнает о том, что он установил соединение именно с google.com, а не с каким-то поддельным сайтом? По умолчанию реализация протокола TLS в Golang задействует конфигурацию вашей операционной системы для выяснения способа аутентификации TLS-серверов. (Подробнее о том, как это происходит, мы поговорим чуть позже.) Процесс использования TLS на стороне сервера также довольно прост, о чем свидетельствует листинг 9.2.

Листинг 9.2. TLS-сервер на языке Golang

```
import (
    "crypto/tls"
    "net/http"
)

func hello(rw http.ResponseWriter, req *http.Request) {
    rw.Write([]byte("Hello, world\n"))
}

func main() {
    config := &tls.Config{
        MinVersion: tls.VersionTLS13,
    }

    http.HandleFunc("/", hello)

    server := &http.Server{
        Addr: ":8080",
        TLSConfig: config,
    }

    cert := "cert.pem"
    key := "key.pem"
    err := server.ListenAndServeTLS(cert, key)
      if err != nil {
        panic(err)
    }
}
```

Минимальная конфигурация для сервера TLS 1.3

Отображает простую страницу с надписью Hello, world

HTTPS-сервер запускается на порте 8080

Несколько файлов .pem, содержащих сертификат и секретный ключ (подробнее об этом — чуть позже)

В данном случае язык Golang и его стандартная библиотека очень многое делают за нас. К сожалению, стандартные библиотеки далеко не всех языков предоставляют простые в использовании реализации протокола TLS, если вообще предоставляют,

и не все библиотеки TLS предусматривают реализации, безопасные по умолчанию! Поэтому в зависимости от библиотеки настройка TLS-сервера может вызвать определенные трудности. В следующем разделе вы узнаете о принципе работы и различных тонкостях протокола TLS.

ПРИМЕЧАНИЕ

Протокол TLS работает поверх TCP. Для защиты UDP-соединений мы можем использовать протокол DTLS (Datagram Transport Layer Security — протокол датаграмм безопасности транспортного уровня; термин «датаграмма» обозначает UDP-сообщения). Данный протокол очень похож на TLS, поэтому я не буду его здесь описывать.

9.2. ПРИНЦИП РАБОТЫ ПРОТОКОЛА TLS

Как уже было сказано, сегодня TLS фактически является стандартным протоколом для защиты коммуникаций между приложениями. В этом разделе вы узнаете о том, как он работает и как используется на практике. Прочитав его, вы научитесь правильно применять протокол TLS, а также поймете принцип работы большинства, если не всех, безопасных транспортных протоколов. Вы также узнаете, почему переделывать или заново реализовывать подобные протоколы так сложно (и категорически не рекомендуется).

На высоком уровне работа протокола TLS делится на две фазы (рис. 9.1).

- *Фаза рукопожатия* — согласование и создание защищенного канала связи между двумя участниками.
- *Фаза передачи данных* — шифрование сообщений, которыми обмениваются участники.

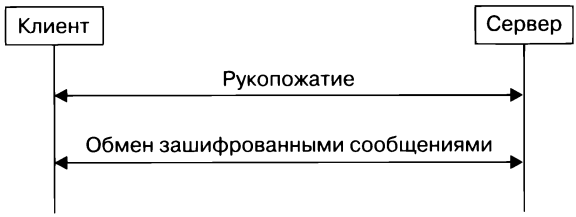


Рис. 9.1. На высоком уровне безопасные транспортные протоколы сначала в фазе рукопожатия создают защищенное соединение. После этого приложения-участники могут безопасно взаимодействовать друг с другом

Благодаря знаниям о гибридном шифровании, полученным в ходе изучения главы 6, вы уже должны знать следующее.

- *По сути, рукопожатие сводится к простому обмену ключами.* В результате рукопожатия два участника достигают договоренности относительно набора симметричных ключей.

- *Фаза передачи данных сводится к шифрованию сообщений, которыми обмениваются участники.* На этом этапе используется алгоритм аутентифицированного шифрования и набор ключей, созданный в результате рукопожатия.

Большинство безопасных транспортных протоколов работают именно так, и самые интересные вещи всегда происходят в фазе рукопожатия, поэтому рассмотрим ее более подробно.

9.2.1. TLS-рукопожатие

Как вы уже видели, TLS, да и большинство безопасных транспортных протоколов, состоит из двух частей: *фазы рукопожатия* и *фазы передачи данных*. В этом разделе мы рассмотрим четыре этапа фазы рукопожатия.

- *Переговоры.* Протокол TLS обладает высокой степенью конфигурируемости. Как клиент, так и сервер могут использовать различные версии протоколов SSL и TLS, а также целый ряд криптографических алгоритмов. Этап переговоров в рамках фазы рукопожатия сводится к поиску точек соприкосновения между конфигурациями клиента и сервера с целью установки безопасного соединения между ними.
- *Обмен ключами.* Основная суть рукопожатия заключается в обмене ключами между двумя участниками. Алгоритм обмена ключами выбирается в ходе переговоров между клиентом и сервером.
- *Аутентификация.* Как было сказано в главе 5, злоумышленники, осуществляющие атаку посредника, могут выдать себя за любую из сторон обмена ключами. В связи с этим обмен ключами требует проведения аутентификации. Ваш браузер должен иметь возможность убедиться в том, что он взаимодействует, например, с сайтом google.com, а не с вашим интернет-провайдером (ISP).
- *Возобновление сеанса.* Поскольку браузеры часто снова и снова устанавливают соединения с одними и теми же веб-сайтами, обмен ключами может требовать больших затрат и замедлять работу пользователя. По этой причине в протокол TLS интегрированы механизмы, позволяющие быстро возобновлять безопасные сеансы связи без повторного обмена ключами.

Теперь подробно рассмотрим каждый пункт этого списка.

Переговоры в TLS — согласование версии и алгоритмов

Значительная часть сложностей при использовании протокола TLS связана с согласованием различных его компонентов. Именно этот этап являлся источником большинства проблем в истории данного протокола. Такие атаки, как FREAK, LOGJAM, DROWN и др., сводились к задействованию слабых мест старых версий с целью взлома более поздних версий протокола (порой даже когда сервер не поддерживал эти старые версии!). Хотя не все протоколы предусматривают версионность и позволяют использовать различные алгоритмы, протокол SSL/TLS был разработан для работы в Интернете, поэтому ему требовался способ обеспечения

обратной совместимости со старыми клиентами и серверами, которые могли обновляться довольно медленно.

Сейчас дела во Всемирной паутине обстоят так: ваш браузер может быть современным и поддерживать протокол TLS версии 1.3, но при посещении какой-нибудь старой веб-страницы есть вероятность того, что сервер, на котором она расположена, поддерживает версии TLS вплоть до 1.2 или 1.1 (или еще хуже). И наоборот, многие веб-сайты вынуждены поддерживать старые браузеры, а значит, и старые версии TLS, поскольку некоторые пользователи застряли в прошлом.

БЕЗОПАСНЫ ЛИ СТАРЫЕ ВЕРСИИ SSL И TLS?

У большинства версий SSL и TLS есть проблемы с безопасностью, за исключением версий TLS 1.2 и 1.3. Почему бы просто не ограничиться поддержкой последней версии (1.3)? Дело в том, что некоторые компании поддерживают старые клиенты, которые не так уж просто обновить. В связи с этим в библиотеках нередко можно встретить реализации средств защиты от известных атак, предназначенные для безопасной поддержки старых версий. К сожалению, эти средства защиты часто оказываются слишком сложными для корректной реализации.

Например, исследователи в области безопасности снова и снова обнаруживали уязвимости для таких известных атак, как Lucky13 и Bleichenbacher98, в различных реализациях TLS, несмотря на попытки исправления соответствующих проблем. Хотя от ряда атак старые версии TLS вполне возможно защитить, я бы не рекомендовал это делать, и я не единственный, кто придерживается этого мнения. В марте 2021 года организация IETF опубликовала документ RFC 8996. Deprecating TLS 1.0 and TLS 1.1, тем самым официально объявив о прекращении поддержки этих версий.

Процесс переговоров начинается с отправки клиентом первого запроса (называемого *ClientHello*) серверу. Этот запрос содержит ряд поддерживаемых версий SSL и TLS, набор криптографических алгоритмов, которые клиент готов использовать, и дополнительную информацию, которая может пригодиться в ходе рукопожатия или оказаться необходимой приложению. Набор криптографических алгоритмов включает:

- *один или несколько алгоритмов обмена ключами.* Версия TLS 1.3 предлагает для этапа переговоров следующие алгоритмы: ECDH с P-256, P-384, P-521, X25519, X448 и FFDH с группами, определенными в документе RFC 7919. Обо всех этих алгоритмах мы говорили в главе 5. Предыдущие версии TLS предлагали также алгоритм обмена ключами RSA (описанный в главе 6), но в последней версии он не используется;
- *два (для разных этапов рукопожатия) и более алгоритма цифровой подписи.* В версии TLS 1.3 применяется алгоритм RSA PKCS#1 версии 1.5 и более новый

RSA-PSS, а также более современные алгоритмы на основе эллиптических кривых, такие как ECDSA и EdDSA, о которых мы говорили в главе 7. Обратите внимание на то, что цифровые подписи создаются с помощью хеш-функции, что позволяет, например, договориться о применении алгоритма RSA-PSS с SHA-256 или SHA-512;

- *одну или несколько хеш-функций для использования с HMAC и HKDF.* Версия TLS 1.3 поддерживает два экземпляра хеш-функции SHA-2 (о которой говорилось в главе 2): SHA-256 и SHA-384. Выбор этой хеш-функции никак не связан с функцией, применяемой алгоритмом цифровой подписи. Напомню, что HMAC — это код аутентификации сообщений, речь о котором шла в главе 3, а HKDF — функция формирования ключа, которую мы рассматривали в главе 8;
- *один или несколько алгоритмов аутентифицированного шифрования.* К ним относятся AES-GCM с ключами длиной 128 или 256 бит, ChaCha20Poly1305 и AES-CCM. Обо всех этих алгоритмах мы говорили в главе 4.

Затем сервер отвечает сообщением *ServerHello*, которое содержит по одному криптографическому алгоритму каждого типа из предложенных клиентом. Этот ответ показан на следующей иллюстрации.



Привет! Я хочу установить соединение с помощью TLS 1.3. Я поддерживаю алгоритмы обмена ключами X448 и X25519, алгоритмы аутентифицированного шифрования AES-GCM и ChaCha20-Poly1305 и т. д.



Привет! Давай использовать алгоритм обмена ключами X25519, алгоритм аутентифицированного шифрования AES-GCM и т. д.

Если сервер не находит поддерживаемый алгоритм, то прерывает соединение. Однако в некоторых случаях вместо этого он может запросить у клиента дополнительную информацию. Для этого сервер отправляет ему сообщение HelloRetryRequest, запрашивая недостающий фрагмент информации. После этого клиент может повторно отправить сообщение ClientHello, содержащее запрошенные сведения.

Протокол TLS и обмен ключами для обеспечения прямой секретности. Обмен ключами — это самая важная часть TLS-рукопожатия, без которой согласование симметричного ключа было бы невозможным. Но для того, чтобы обмен ключами состоялся, клиент и сервер должны сначала обменяться своими открытыми ключами.

В TLS 1.2 и более ранних версиях клиент и сервер начинают процесс обмена ключами только после согласования соответствующего алгоритма на этапе переговоров. Версия TLS 1.3 оптимизирует этот процесс, позволяя проводить переговоры и обмениваться ключами одновременно, при этом клиент выбирает один из поддерживаемых алгоритмов обмена ключами и отправляет открытый ключ в первом сообщении (ClientHello). Если выбранный им алгоритм обмена ключами не совпадает с выбором сервера, то он обращается к результатам переговоров и отправляет новое сообщение ClientHello, содержащее корректный открытый ключ. Это может выглядеть следующим образом (разница между версиями TLS 1.2 и TLS 1.3 показана на рис. 9.2).

1. Клиент отправляет сообщение TLS 1.3 ClientHello, в котором говорится о том, что он может выполнить обмен ключами с использованием алгоритма X25519 или X448. В этом же сообщении он отправляет открытый ключ X25519.
2. Сервер не поддерживает алгоритм X25519, но поддерживает X448. Он отправляет клиенту сообщение HelloRetryRequest, в котором говорится, что он поддерживает только алгоритм X448.
3. Клиент отправляет то же сообщение ClientHello, но на этот раз с открытым ключом X448.
4. Процесс рукопожатия продолжается.

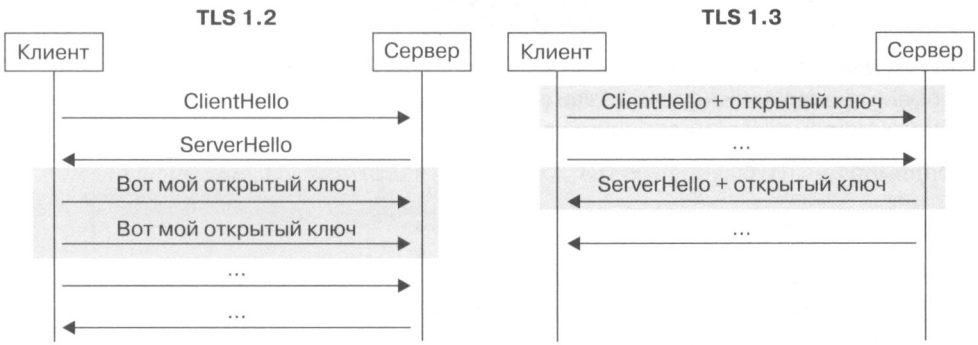


Рис. 9.2. В версии TLS 1.2 клиент ждет, пока сервер выберет алгоритм обмена ключами, прежде чем отправить открытый ключ. В версии TLS 1.3 клиент делает предположение относительно того, какой (-ие) алгоритм (-ы) обмена ключами выберет сервер, и отправляет открытый ключ (или несколько) в первом сообщении, что иногда позволяет избежать лишнего раунда обмена сообщениями

Версия TLS 1.3 полна подобных оптимизаций, которые имеют очень большое значение для пользователей Интернета. Дело в том, что у многих жителей планеты интернет-соединение весьма нестабильное или медленное, что требует сведения к минимуму коммуникаций, которые не имеют непосредственного отношения к работе приложений. Кроме того, в TLS 1.3, в отличие от предыдущих версий данного протокола, все эпизоды обмена ключами являются *эфемерными*. Это означает, что для каждого нового сеанса клиент и сервер генерируют новые

пары ключей, а после обмена ключами избавляются от них. Это обеспечивает так называемую *прямую секретность* или *секретность в будущем*, благодаря которой компрометация долгосрочных ключей клиента или сервера не позволяет злоумышленнику расшифровать этот сеанс при условии безопасного удаления эфемерных закрытых ключей.

Представьте, что произойдет, если вместо этого TLS-сервер будет использовать один закрытый ключ в ходе каждого обмена ключами с клиентами. Обмениваясь эфемерными ключами и избавляясь от закрытых ключей сразу после завершения рукопожатия, сервер защищается от подобных злоумышленников. Суть данной концепции проиллюстрирована на рис. 9.3.

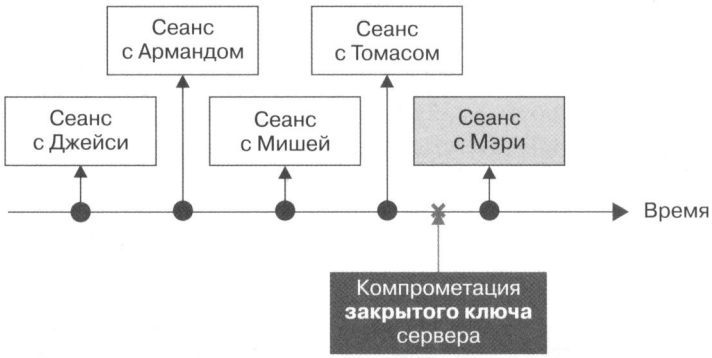


Рис. 9.3. В версии протокола TLS 1.3 каждый сеанс начинается с обмена эфемерными ключами. Если в какой-то момент времени сервер будет скомпрометирован, это никак не повлияет на предыдущие сеансы

УПРАЖНЕНИЕ

Компрометация закрытого ключа сервера в определенный момент будет иметь катастрофические последствия, поскольку злоумышленники, осуществляющие атаки посредника, смогут расшифровать все ранее записанные разговоры. Вы понимаете, как это может произойти?

После обмена эфемерными открытыми ключами происходит формирование ключей и обмен ими. В версии TLS 1.3 разные ключи формируются в разные моменты времени для шифрования различных этапов с помощью независимых ключей.

Первые два сообщения, ClientHello и ServerHello, не могут быть зашифрованы, поскольку к этому моменту обмен открытыми ключами еще не состоялся. Однако после обмена ключами TLS 1.3 шифрует остальные этапы рукопожатия. (В этом заключается отличие от предыдущих версий TLS, которые не шифровали ни одно из сообщений, отправляемых в ходе рукопожатия.)

Для формирования различных ключей в TLS 1.3 используется алгоритм HKDF с согласованной хеш-функцией. Функция HKDF-Extract применяется

к выводу процесса обмена ключами для удаления любых смещений, а функция HKDF-Expand задействуется с различными параметрами `info` для формирования ключей шифрования. Например, код `tls13 c hs traffic` (клиентский трафик рукопожатия) применяется для формирования симметричных ключей, предназначенных для шифрования данных, отправляемых клиентом серверу в ходе рукопожатия, а `tls13 s ap traffic` (трафик серверного приложения) — для получения симметричных ключей, предназначенных для шифрования данных, отправляемых сервером клиенту после рукопожатия. Однако помните о том, что *неаутентифицированный* обмен ключами небезопасен! Далее мы поговорим о том, как TLS решает эту проблему.

Аутентификация в TLS и веб-инфраструктура открытых ключей. После переговоров и обмена ключами выполняется другая важная часть процесса TLS-рукопожатия, называемая *аутентификацией*. В главе 5 вы видели, насколько легко можно перехватить данные и выдать себя за одну или обе стороны обмена ключами. В данном разделе я объясню, как ваш браузер криптографически устанавливает подлинность сайта, с которым взаимодействует. Однако перед этим сделаем шаг назад, поскольку есть еще кое-что, о чем я вам не рассказал. На самом деле в версии TLS 1.3 процесс рукопожатия состоит из трех этапов (рис. 9.4).

1. *Обмен ключами*. На данном этапе отправляются сообщения ClientHello и ServerHello, с помощью которых стороны проводят переговоры и обмениваются ключами. Все сообщения, включая сообщения рукопожатия, отправляемые после завершения этого этапа, шифруются.
2. *Параметры сервера*. Сообщения, пересылаемые на этом этапе, включают дополнительные данные сервера, которые не обязательно должны содержаться в его первом сообщении и могут требовать шифрования.
3. *Аутентификация*. На этом этапе сервер и клиент обмениваются информацией, необходимой для их аутентификации.



Рис. 9.4. В версии TLS 1.3 процесс рукопожатия состоит из трех этапов: обмена ключами, передачи параметров сервера и аутентификации

В веб-контексте TLS-аутентификация обычно односторонняя. Только браузер проверяет подлинность сайта, например, google.com, однако сам сайт не выполняет такую проверку (или по крайней мере не делает этого в рамках TLS-рукопожатия).

TLS С ВЗАИМНОЙ АУТЕНТИФИКАЦИЕЙ

В Интернете аутентификация клиента часто выполняется на прикладном уровне, как правило, с помощью формы, запрашивающей его учетные данные. Однако она может происходить и в рамках TLS-рукопожатия, если будет запрошена сервером на этапе передачи его параметров. Когда обе стороны соединения проходят аутентификацию, речь идет о протоколе *TLS с взаимной аутентификацией*, который иногда сокращенно называется mTLS.

Аутентификация клиента выполняется так же, как и аутентификация сервера. Она может произойти в любой момент после аутентификации сервера, например во время рукопожатия или после его завершения.

Теперь давайте узнаем о том, как браузер проверяет подлинность сайта, к которому подключается. Для этого он использует *веб-инфраструктуру открытых ключей* (web public key infrastructure, web PKI), далее веб-ИОК.

Мы познакомились с понятием инфраструктуры открытых ключей в главе 7, посвященной цифровым подписям, однако позвольте мне еще раз вкратце рассказать об этой концепции, поскольку она очень важна для понимания принципа работы Всемирной паутины. Веб-ИОК имеет два аспекта. Во-первых, браузеры должны доверять наборам корневых открытых ключей, которые называются *центрами сертификации* (ЦС) или *удостоверяющими центрами* (УЦ). Обычно браузеры либо используют жестко закодированный набор доверенных открытых ключей, либо полагаются на то, что их предоставит операционная система.

ВЕБ-ИОК

Во Всемирной паутине существуют сотни независимых центров сертификации, которыми управляют различные компании и организации, работающие по всему миру. Эту систему довольно сложно анализировать, и сами ЦС иногда могут подписывать открытые ключи промежуточных ЦС, которые, в свою очередь, имеют право подписывать открытые ключи веб-сайтов. По этой причине такие организации, как *Форум центров сертификации и разработчиков браузеров* (Certification Authority Browser Forum, CA/Browser Forum), устанавливают правила и решают, когда новые организации могут присоединиться к набору доверенных открытых ключей или когда тому или иному ЦС больше нельзя доверять и он должен быть исключен из этого набора.

Во-вторых, сайты, которые хотят использовать протокол HTTPS, должны иметь возможность получить сертификат (подпись их открытого ключа электронной подписи) от этих центров сертификации. Для этого владелец сайта (или веб-мастер) должен доказать ЦС, что он — владелец конкретного домена.

ПРИМЕЧАНИЕ

Раньше за получение сертификата для своего сайта приходилось платить. Теперь это не так, поскольку такие ЦС, как Let's Encrypt, предоставляют сертификаты бесплатно.

Например, чтобы удостовериться в том, что вы владеете сайтом example.com, центр сертификации может попросить вас разместить по адресу example.com/some_path/file.txt файл, содержащий несколько случайных чисел, сгенерированных для вашего запроса. Этот процесс проиллюстрирован на следующем рисунке.



После этого ЦС может подписать открытый ключ веб-сайта. Поскольку подпись ЦС обычно остается действительной на протяжении нескольких лет, в данном случае речь идет о долгосрочном открытом ключе электронной подписи (в противоположность эфемерному открытому ключу). Если более конкретно, то центры сертификации на самом деле подписывают не открытые ключи, а *сертификаты* (подробнее об этом — чуть позже). Сертификат содержит долгосрочный открытый ключ и дополнительные важные метаданные, например доменное имя веб-страницы.

Чтобы доказать вашему браузеру, что он действительно взаимодействует с сервером google.com, этот сервер в рамках TLS-рукопожатия отправляет *цепочку сертификатов*. Она включает в себя:

- собственный листовой сертификат, содержащий, помимо прочего, доменное имя (например, google.com), долгосрочный открытый ключ Google, а также подпись центра сертификации;
- цепочку сертификатов промежуточных ЦС, начиная с того, который подписал сертификат Google, и заканчивая корневым ЦС, подписавшим сертификат последнего из промежуточных центров сертификации.

Это весьма многословное описание проиллюстрировано на рис. 9.5.

Цепочка сертификатов отправляется сервером и клиентом в специальном TLS-сообщении так, будто клиенту предлагается пройти аутентификацию. После этого сервер может использовать свою сертифицированную пару долгосрочных ключей для подписания всех сообщений рукопожатия, полученных и ранее отправленных в так называемом сообщении *CertificateVerify*. Этот процесс, в рамках которого аутентификацию проходит только сервер, проиллюстрирован на рис. 9.6.

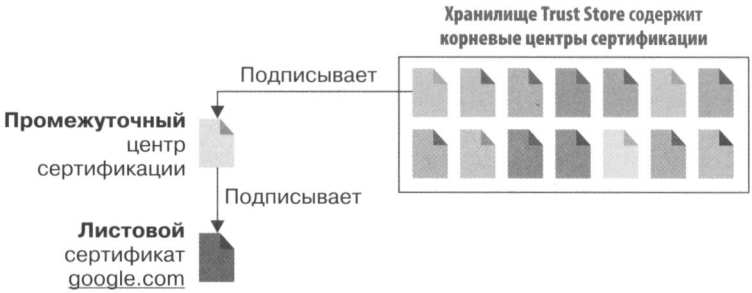


Рис. 9.5. Для того чтобы доверять Всемирной паутине в целом, веб-браузерам достаточно доверять относительно небольшому набору корневых центров сертификации, которые находятся в хранилище Trust Store. Чтобы браузер мог доверять конкретному веб-сайту, его листовой сертификат должен быть подписан одним из этих ЦС. Иногда корневые центры сертификации подписывают только сертификаты промежуточных ЦС, которые, в свою очередь, подписывают другие промежуточные или листовые сертификаты. Эта система известна как веб-ИОК

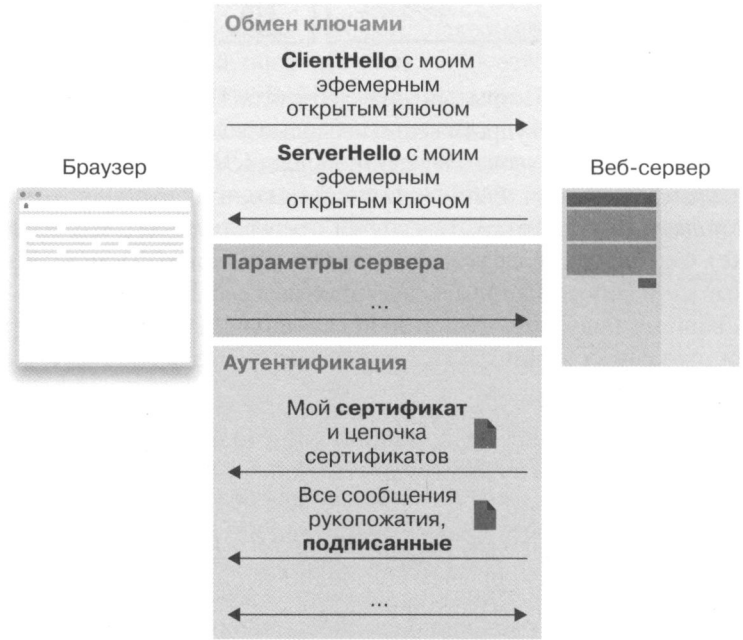


Рис. 9.6. Аутентификация в рамках рукопожатия начинается с того, что сервер отправляет клиенту цепочку сертификатов, которая начинается с листового сертификата (он содержит открытый ключ веб-сайта и дополнительные метаданные, например имя домена) и заканчивается корневым сертификатом, которому доверяет браузер. Каждый сертификат содержит подпись сертификата, расположенного выше по цепочке

Подпись в сообщении CertificateVerify удостоверяет все то, что сервер видел до этого. Без данной подписи злоумышленник, реализующий атаку посредника, мог бы перехватить сообщения рукопожатия сервера, заменить его эфемерный открытый ключ, содержащийся в сообщении ServerHello, и успешно выдать себя за этот сервер. Подумайте, почему злоумышленник не может заменить эфемерный открытый ключ сервера при наличии подписи CertificateVerify.

ВРЕМЯ ДЛЯ РАССКАЗА

Несколько лет назад меня наняли для проверки протокола TLS, разработанного одной крупной компанией. Оказалось, что в рамках этого протокола сервер предоставлял подпись, которая не защищала эфемерный ключ. Когда я рассказал об этой проблеме, все присутствующие в комнате замолчали на целую минуту. Это была весьма существенная ошибка, поскольку злоумышленник, способный перехватить сообщения рукопожатия и заменить эфемерный ключ своим собственным, мог успешно выдать себя за сервер.

Урок в данном случае заключается в том, что не стоит изобретать велосипед. Безопасные транспортные протоколы трудно реализовать корректно, и, как показывает история, они могут давать сбои самым неожиданным образом. Вместо этого следует полагаться на зрелые протоколы, такие как TLS, и использовать их популярную реализацию, которая уже получила значительную долю общественного внимания.

Наконец, для официального завершения процесса рукопожатия обе стороны соединения должны в рамках этапа аутентификации отправить сообщение *Finished*. Оно содержит метку аутентификации, созданную механизмом HMAC, инстанцированную хеш-функцией, согласованной для данного сеанса. Это позволяет клиенту и серверу сообщить другой стороне: «Вот все сообщения, которые я отправил и получил в порядке очереди в ходе этого рукопожатия». Если рукопожатие было перехвачено и подделано злоумышленниками, реализующими атаку посредника, эта проверка целостности позволяет участникам обнаружить данный факт и прервать соединение. Это особенно полезно, поскольку некоторые режимы рукопожатий *не* предусматривают использование подписи (подробнее об этом — чуть позже).

Прежде чем переходить к следующему аспекту рукопожатия, рассмотрим сертификаты X.509 — важный компонент многих криптографических протоколов.

Аутентификация с помощью сертификатов X.509. Хотя в версии TLS 1.3 сертификаты необязательны, так как вы всегда можете применять обычные ключи, многие приложения и протоколы (не только веб) активно используют их для удостоверения дополнительных метаданных, в частности стандарт сертификатов X.509 версии 3.

X.509 — это довольно старый стандарт, который должен был быть достаточно гибким для множества сценариев: от электронной почты до веб-страниц. Стандарт X.509 использует так называемый *язык описания абстрактного синтаксиса данных* (Abstract Syntax Notation One, ASN.1) для указания информации, содержащейся в сертификате. Структура данных, описанная с помощью ASN.1, выглядит следующим образом:

```
Certificate  ::=  SEQUENCE  {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING }
```

Вы можете буквально прочитать этот фрагмент как структуру, содержащую три поля:

- `tbsCertificate` — подлежащий подписанию сертификат, в котором содержится вся информация, требующая сертификации. В веб-контексте это может быть доменное имя (например, google.com), открытый ключ, дата истечения срока действия и т. д.;
- `signatureAlgorithm` — алгоритм, используемый для подписи сертификата;
- `signatureValue` — подпись центра сертификации.

УПРАЖНЕНИЕ

Значения `signatureAlgorithm` и `signatureValue` не содержатся в самом сертификате `tbsCertificate`. Знаете ли вы почему?

Вы можете легко проверить содержимое сертификата X.509, подключившись к любому веб-сайту с помощью HTTPS и используя функции браузера для просмотра цепочки сертификатов, отправленной сервером (рис. 9.7).

Вы можете встретить сертификаты X.509 в виде файлов .pem, которые представляют собой некоторое содержимое в кодировке base64, окруженное понятным человеку намеком на то, что содержат закодированные данные (здесь — сертификат). Следующий фрагмент представляет собой содержимое сертификата в формате .pem:

```
-----BEGIN CERTIFICATE-----
MIIJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoXDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
[...]
vaoUqelfNJJvQjJbMQbSQEp9y8EIi4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRiplKVzZSnUvypgBLryYnl6kquh1AJS5gnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6thTjwPnfB5aVNK
dl0nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW6l/vt3coyyDkQKDmT61tjwy5k=
-----END CERTIFICATE-----
```

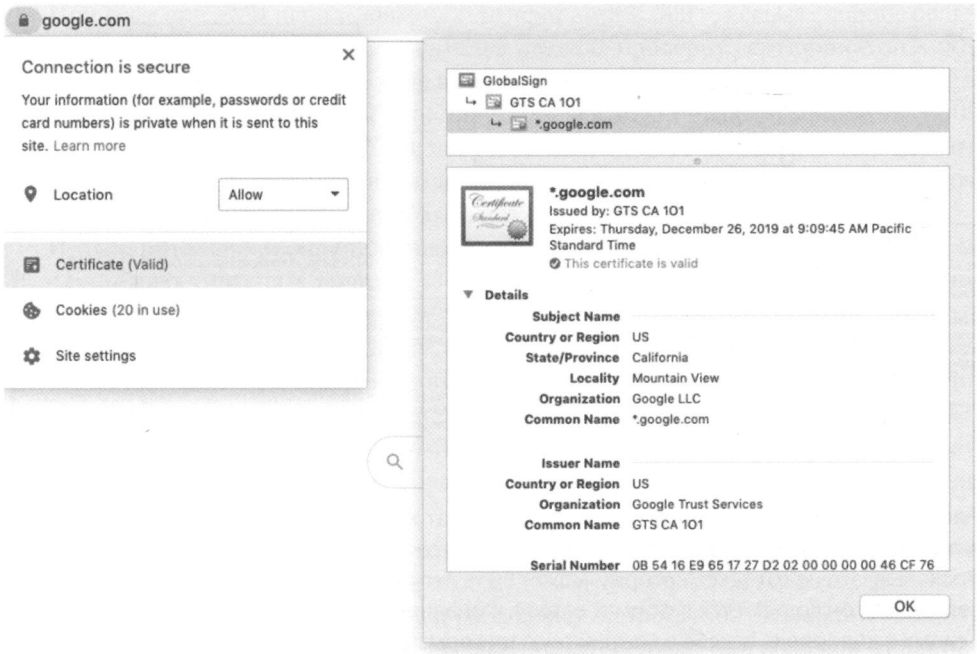


Рис. 9.7. С помощью программы просмотра сертификатов браузера Chrome мы можем просмотреть цепочку сертификатов, отправленную сервером Google. Корневым центром сертификации является Global Sign, которому доверяет ваш браузер. Ниже по цепочке находится промежуточный ЦС под названием GTS CA 101, который является доверенным благодаря тому, что его сертификат содержит подпись Global Sign. В свою очередь, листовой сертификат Google, действительный для *.google.com (google.com, mail.google.com и т. д.), содержит подпись центра GTS CA 101

Если вы расшифруете содержимое в кодировке base64, окруженное фразами `BEGIN CERTIFICATE` и `END CERTIFICATE`, то получите сертификат, закодированный в соответствии с *отличительными правилами кодирования* (distinguished encoding rules, DER). DER представляет собой *детерминированную* (допускающую только один способ кодирования) двоичную кодировку, используемую для преобразования сертификатов X.509 в байты. Подобные кодировки часто сбивают новичков с толку! Я проиллюстрировал их все на рис. 9.8.

При использовании правил DER информация кодируется в виде «здесь целое число» или «это байтовый массив». В результате такого кодирования теряются имена полей, описанные с помощью ASN.1, например `tbsCertificate`. Таким образом, декодирование данных, закодированных по правилам DER, без знания исходного описания каждого поля на языке ASN.1 бессмысленно. Инструменты командной строки наподобие OpenSSL позволяют декодировать и перевести на человеческий язык содержимое сертификата, закодированного в формате DER. Например, если

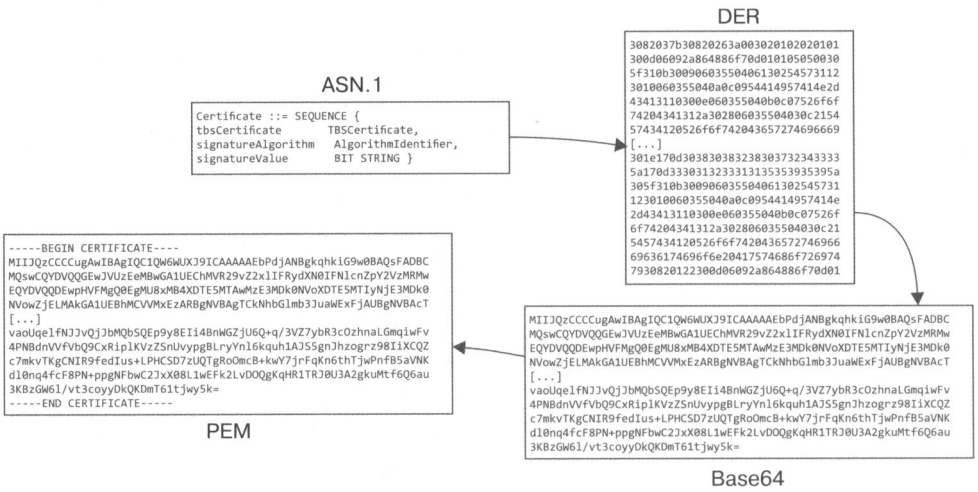


Рис. 9.8. В левом верхнем углу показан сертификат X.509, записанный с помощью нотации ASN.1. Затем он преобразуется в байты в соответствии с правилами DER. Поскольку это не тот текст, который может быть легко скопирован или распознан человеком, он кодируется в формат base64. Финальным штрихом является обертывание данных в кодировке base64 в контекстную информацию с помощью формата .pem

вы загрузите сертификат google.com, то сможете использовать следующий фрагмент кода для отображения его содержимого в терминале:

```
$ openssl x509 -in google.pem -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0b:54:16:e9:65:17:27:d2:02:00:00:00:00:46:cf:76
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Google Trust Services, CN = GTS CA 1O1
        Validity
            Not Before: Oct  3 17:09:45 2019 GMT
            Not After : Dec 26 17:09:45 2019 GMT
        Subject: C = US, ST = California, L = Mountain View, O = Google LLC,
                 CN = *.google.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:74:25:79:7d:6f:77:e4:7e:af:fb:1a:eb:4d:41:
                    b5:27:10:4a:9e:b8:a2:8c:83:ee:d2:0f:12:7f:d1:
                    77:a7:0f:79:fe:4b:cb:b7:ed:c6:94:4a:b2:6d:40:
                    5c:31:68:18:b6:df:ba:35:e7:f3:7e:af:39:2d:5b:
                    43:2d:48:0a:54
                ASN1 OID: prime256v1
                NIST CURVE: P-256
[...]
```

При всем при этом использование сертификатов X.509 — весьма спорное решение. В 2012 году команда исследователей назвала проверку сертификатов X.509 применением самого опасного кода в мире. Это связано с тем, что кодировка DER тяжело поддается корректному анализу, а из-за сложности сертификатов X.509 многие потенциальные ошибки могут оказаться весьма разрушительными. По этой причине я не рекомендую использовать сертификаты X.509 ни в одном современном приложении, если в этом нет особой необходимости.

Предварительно распределенные ключи и возобновление сеанса в TLS, или Как избежать обмена ключами. Обмен ключами может быть дорогостоящим, а иногда и совершенно ненужным. Например, у вас есть две машины, которые подключаются только друг к другу, и вы не хотите задействовать инфраструктуру открытых ключей для защиты их коммуникаций. Версия протокола TLS 1.3 позволяет избежать этих накладных расходов благодаря так называемым *предварительно распределенным ключам* (pre-shared keys, PSK). PSK — это секрет, известный клиенту и серверу, который может использоваться при формировании симметричных ключей для сеанса.

В версии TLS 1.3 PSK-рукопожатие работает следующим образом: в своем сообщении ClientHello клиент сообщает о том, что он поддерживает некоторый список PSK-идентификаторов. Если сервер распознает один из них, он может сообщить об этом в ответном сообщении ServerHello, и в этом случае они оба (при желании) смогут избежать обмена ключами. При этом пропускается этап аутентификации, что делает сообщение Finished в конце рукопожатия важным для предотвращения реализации атаки посредника.

СЛУЧАЙНОЕ ЗНАЧЕНИЕ КЛИЕНТА И СЛУЧАЙНОЕ ЗНАЧЕНИЕ СЕРВЕРА

Внимательный читатель мог заметить, что эфемерные открытые ключи привносят в сеанс элемент случайности, и без них симметричные сеансовые ключи в конце рукопожатия могут всегда оказываться одинаковыми. Использование разных симметричных ключей для разных сеансов очень важно, поскольку мы не хотим, чтобы эти сеансы были связаны между собой. Более того, возможные различия в зашифрованных сообщениях между сеансами могут привести к повторному использованию nonce, имеющему катастрофические последствия (см. главу 4).

Чтобы смягчить эту проблему, в сообщениях ClientHello и ServerHello предусмотрено поле random, значение которого генерируется случайным образом для каждого нового сеанса и часто называется *случайным значением клиента* (client random) и *случайным значением сервера* (server random). Поскольку эти случайные значения применяются при формировании симметричных ключей в рамках протокола TLS, это фактически приводит к рандомизации симметричных сеансовых ключей для каждого нового соединения.

Помимо этого, PSK задействуются для *возобновления сеанса*, которое предполагает повторное использование секретов, созданных в ходе предыдущего сеанса или соединения. Если вы уже подключились к сайту google.com, проверили его цепочку сертификатов, обменялись ключами, согласовали общий секрет и т. д., зачем повторять все это спустя несколько минут или часов при повторном посещении этого сайта? Версия TLS 1.3 позволяет после успешного выполнения рукопожатия генерировать PSK, который можно использовать в ходе последующих соединений, чтобы избежать повторной реализации полноценного процесса рукопожатия.

Если сервер хочет предложить эту возможность, он может отправить сообщение *New Session Ticket* в любое время после завершения рукопожатия. Сервер может создавать так называемые *сеансовые мандаты* (session ticket) несколькими способами. Например, он может отправить идентификатор, связанный с соответствующей информацией в базе данных. Это не единственный способ, но, поскольку он довольно сложен и чаще всего неактуален, я не буду больше упоминать о нем в этой главе. Далее мы рассмотрим самую простую часть протокола TLS, а именно процесс шифрования сообщений.

9.2.2. Шифрование данных приложения с помощью протокола TLS 1.3

После выполнения рукопожатия и формирования симметричных ключей клиент и сервер могут начать обмениваться зашифрованными данными приложения. Но это еще не все: протокол TLS также гарантирует, что такие сообщения не могут быть повторно переданы или переупорядочены! С этой целью алгоритм аутентифицированного шифрования использует nonce, которое начинается с фиксированного значения и инкрементируется при шифровании каждого нового сообщения. В случае повторной передачи или переупорядочения сообщений значение nonce будет отличаться от ожидаемого и расшифровка не удастся. Когда такое происходит, соединение прерывается.

СОКРЫТИЕ ДЛИНЫ ОТКРЫТОГО ТЕКСТА

Как вы узнали в главе 4, шифрование не всегда позволяет скрыть длину шифруемого сообщения. Версия TLS 1.3 предусматривает функцию *дополнения записей* (record padding), с помощью которой перед шифрованием данных приложения в них можно вносить случайное число нулевых байтов, скрывая тем самым истинную длину сообщения. Несмотря на это, существуют атаки, устраняющие внесенный в данные шум, и защититься от них не так просто. Если вам действительно необходимо использовать это свойство безопасности, обратитесь к спецификации протокола TLS 1.3.

Начиная с версии TLS 1.3 при наличии соответствующего разрешения сервера клиенты могут отправлять зашифрованные данные в составе первой серии сообщений, следующих сразу после сообщения ClientHello. Это означает, что браузеру

не обязательно дожидаться окончания процесса рукопожатия, чтобы начать отправлять данные приложения на сервер. Этот механизм называется передачей *ранних данных*, или *0-RTT* (zero round trip time — нулевое время возобновления приема/передачи). Его можно использовать только в сочетании с PSK, поскольку он позволяет формировать симметричные ключи во время передачи сообщения ClientHello.

ПРИМЕЧАНИЕ

Эта функция вызвала много споров во время разработки стандарта TLS 1.3, поскольку пассивный злоумышленник может повторно передать наблюдаемое сообщение ClientHello, за которым следуют зашифрованные данные 0-RTT. Поэтому механизм 0-RTT нужно использовать только с теми данными приложения, повторная передача которых безопасна.

Интернет-браузеры рассматривают каждый GET-запрос как *идемпотентный*. Это означает, что GET-запросы не должны менять состояние на стороне сервера и предназначены только для получения данных, в отличие, например, от POST-запросов. Разумеется, так бывает не всегда, и в некоторых случаях приложения делают все, что им вздумается. Поэтому, если вы стоите перед выбором, использовать механизм 0-RTT или нет, имейте в виду, что гораздо проще этого не делать.

9.3. СОВРЕМЕННОЕ СОСТОЯНИЕ ЗАШИФРОВАННОГО ВЕБА

Нынешние стандарты призывают к отказу от использования всех версий SSL и TLS, кроме TLS 1.2 и TLS 1.3. Тем не менее из-за применения устаревших клиентов и серверов многие библиотеки и приложения продолжают поддерживать старые версии протокола (иногда вплоть до SSL версии 3!). Это непросто и требует применения множества труднореализуемых средств защиты из-за большого количества уязвимостей.

ВНИМАНИЕ

Протокол TLS 1.3 (и TLS 1.2) безопасный, и его применение считается лучшей практикой. Использование более старой версии требует консультации экспертов и продумывания способов защиты от известных уязвимостей.

По умолчанию браузеры до сих пор подключаются к веб-серверам по протоколу HTTP, а веб-сайты по-прежнему вынуждены вручную запрашивать сертификат у ЦС. Это означает, что при использовании нынешних протоколов Всемирная паутина никогда не будет полностью зашифрована, хотя, по некоторым оценкам, в 2019 году глобальный веб-трафик был зашифрованным на 90 %.

Проблемой является и то, что по умолчанию ваш браузер всегда задействует небезопасное соединение. Сейчас веб-серверы обычно переключают пользователей, заходящих на их страницы по протоколу HTTP, на применение HTTPS.

Кроме того, веб-серверы могут и часто приказывают браузерам задействовать HTTPS для последующих соединений. Это делается с помощью заголовка HTTPS-ответа, называемого *HTTP Strict Transport Security* (HSTS). Тем не менее первое подключение к веб-сайту остается незащищенным, если только сам пользователь не вводит в адресной строке `https`, и может быть перехвачено, чтобы предотвратить переключение на протокол HTTPS.

Кроме того, другие веб-протоколы, такие как *NTP* (для получения текущего времени) и *DNS* (для получения IP-адреса, связанного с определенным доменным именем), в настоящее время практически не подвергаются шифрованию и уязвимы для атаки посредника. Несмотря на проведение исследований, направленных на улучшение существующего положения дел, это те векторы атак, о которых необходимо знать.

Еще одной угрозой для пользователей протокола TLS оказываются недобросовестные центры сертификации. Что, если сегодня какой-нибудь ЦС решит подписать сертификат для вашего домена и открытый ключ, который он контролирует? Если он сумеет занять позицию для реализации атаки посредника, то сможет выдавать себя за ваш сайт. Если вы контролируете клиентскую сторону соединения, то очевидное решение заключается в том, чтобы не задействовать веб-ИОК (и задействовать собственную инфраструктуру открытых ключей) либо выполнить *привязку* конкретного сертификата или открытого ключа.

Привязка сертификата или открытого ключа — это техника, при использовании которой сертификат сервера (а часто и его хеш) или открытый ключ жестко прописываются непосредственно в коде клиента. Если сервер не предоставляет ожидаемый сертификат или последний не содержит ожидаемого долгосрочного открытого ключа, клиент прерывает соединение на этапе аутентификации в ходе рукопожатия. Такая практика часто используется в мобильных приложениях, поскольку они точно знают, как должен выглядеть открытый ключ или сертификат сервера (в отличие от браузеров, которым приходится подключаться к огромному количеству серверов). Однако жесткое прописывание сертификатов и открытых ключей в коде не всегда возможно, поэтому для борьбы с «плохими» сертификатами существует два других механизма:

- *отзыв сертификата*. Как следует из названия, этот механизм позволяет ЦС отозвать сертификат и предупредить об этом браузеры;
- *мониторинг сертификатов*. Это относительно новый механизм, который заставляет центры сертификации публично регистрировать каждый подписанный сертификат.

История с отзывом сертификатов развивалась довольно неровно. Первым из предложенных решений было использование списков *CRL* (certificate revocation lists), в которых содержались отозванные СЦ сертификаты, то есть сертификаты, которые больше не считались действительными. Однако проблемой этих списков является то, что они могут стать довольно большими и их нужно постоянно проверять.

Впоследствии от списков CRL отказались в пользу *протокола состояния сетевого сертификата* (online certificate status protocol, OCSP), позволяющего проверить

статус отзыва сертификата путем отправки запроса простым веб-интерфейсам. У данного протокола есть проблемы: он требует от ЦС наличия высокодоступной службы, способной отвечать на запросы OCSP, он сливает информацию о веб-трафике центрам сертификации, а браузеры часто решают игнорировать запросы OCSP с истекшим временем ожидания, чтобы не мешать работе пользователя. Современное решение заключается в применении расширения, называемого *сшивкой OCSP* (OCSP stapling). При этом веб-сайт запрашивает у центра сертификации подписанный статус своего сертификата и прикрепляет ответ к этому сертификату в ходе TLS-рукопожатия. Все три варианта решения показаны на рис. 9.9.

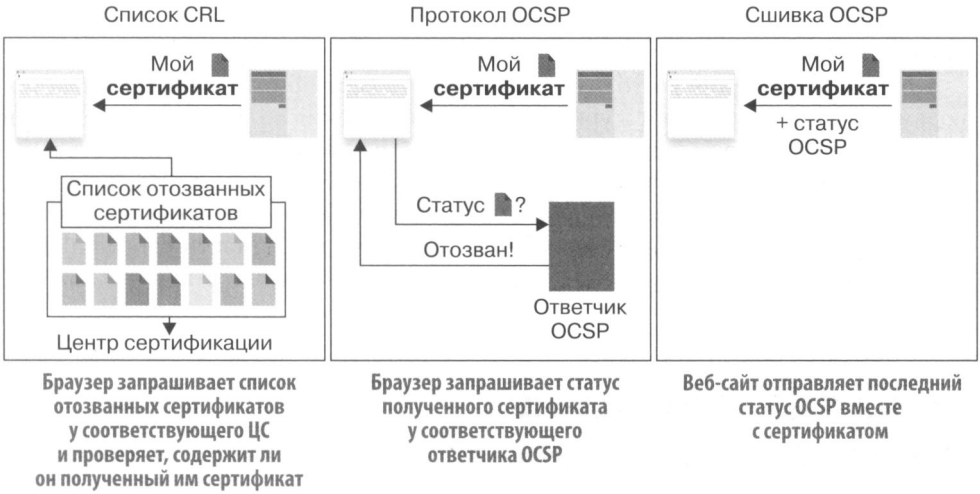


Рис. 9.9. Для отзыва сертификата в Интернете использовались три популярных решения: списки отозванных сертификатов (CRL), протокол состояния сетевого сертификата (OCSP) и сшивка OCSP

Отзыв сертификата может казаться не самой важной функцией (особенно если речь идет о небольших системах, а не о Всемирной паутине), но лишь до тех пор, пока сертификат не будет скомпрометирован. Подобно автомобильному ремню безопасности, отзыв сертификата представляет собой функцию безопасности, которая большую часть времени бесполезна, но в редких случаях может стать настоящим спасением. Это то, что специалисты по безопасности называют глубоко эшелонированной защитой.

ПРИМЕЧАНИЕ

В веб-контексте отзыв сертификата в большинстве случаев оказывается правильным решением. Выявленная в 2014 году ошибка Heartbleed стала одной из самых разрушительных в истории протоколов SSL и TLS. В наиболее распространенной реализации SSL/TLS (OpenSSL) была обнаружена ошибка *чтения за пределами буфера*, то есть чтения за пределами массива, позволявшая любому человеку отправить специально изготовленное сообщение любому серверу OpenSSL и получить дамп его памяти, часто содержащий его долгосрочные закрытые ключи.

Тем не менее, если ЦС действительно ведет себя злонамеренно, он может не отзывать вредоносные сертификаты или не сообщать о них. Проблема в следующем: мы слепо полагаемся на то, что большое число акторов (ЦС) будут поступать правильно. В качестве масштабного решения этой проблемы в 2012 году компания Google предложила идею *прозрачности сертификатов* (certificate transparency), которая заключается в том, чтобы заставить центры сертификации регистрировать каждый выданный ими сертификат в специальном общедоступном журнале. Теперь браузеры, такие как Chrome, отклоняют сертификаты, если они не содержат доказательств своей регистрации в публичном журнале. Прозрачность позволяет проверить факт ошибочной выдачи сертификата принадлежащему вам домену (в журнале не должно быть других сертификатов, кроме тех, которые вы запрашивали в прошлом).

Обратите внимание на то, что прозрачность сертификатов обеспечивается людьми, которые проверяют журналы с целью выявления «плохих» сертификатов для своего домена *постфактум*. ЦС также должны оперативно реагировать и отзывать ошибочно выданные сертификаты после выявления этого факта. В крайних случаях браузеры иногда исключают недобросовестные центры сертификации из своих хранилищ Trust Store. Таким образом, идея прозрачности сертификатов менее эффективна по сравнению с привязкой сертификатов или открытых ключей, которая позволяет смягчить последствия недобросовестных действий ЦС.

9.4. ДРУГИЕ БЕЗОПАСНЫЕ ТРАНСПОРТНЫЕ ПРОТОКОЛЫ

Итак, теперь вы знаете о TLS — самом популярном протоколе для шифрования сообщений. Однако это еще не все. TLS — это не единственный безопасный транспортный протокол в своем классе. Существует множество других, и вы, скорее всего, уже используете какой-то из этих протоколов. Однако большинство из них представляют собой TLS-подобный протокол, адаптированный для конкретного случая применения. Так, например, обстоит дело со следующими протоколами:

- *Secure Shell* (SSH) — это наиболее широко используемый протокол и приложение для безопасного подключения к удаленному терминалу на другой машине;
- *Wi-Fi Protected Access* (WPA) — самый популярный протокол для подключения устройств к точкам доступа частной сети или к Интернету;
- *IPSec* — один из самых популярных протоколов виртуальных сетей (VPN) для соединения различных частных сетей между собой. В основном он используется компаниями для соединения различных офисных сетей. Как следует из названия, он действует на уровне IP и часто задействуется маршрутизаторами, брандмауэрами и прочими сетевыми устройствами. Другой популярный VPN-протокол — OpenVPN, который напрямую использует TLS.

Все эти протоколы, как правило, реализуют парадигму рукопожатия/передачи данных, привнося в нее несколько нюансов. Однако подобное изобретение колеса

не обходится без проблем. Например, несколько Wi-Fi-протоколов были взломаны. В завершение этой главы я хочу познакомить вас с *протокольным фреймворком Noise* (Noise protocol framework), который представляет собой гораздо более современную альтернативу протоколу TLS.

9.5. ПРОТОКОЛЬНЫЙ ФРЕЙМВОРК NOISE — СОВРЕМЕННАЯ АЛЬТЕРНАТИВА TLS

Протокол TLS уже достаточно зрелый и в большинстве случаев считается надежным решением благодаря тому, что ему уделяется много внимания. Тем не менее использующие TLS приложения сталкиваются с дополнительными накладными расходами, обусловленными историческими причинами, ограничениями, связанными с обратной совместимостью, и общей сложностью. Действительно, во многих сценариях, где вы контролируете все конечные точки, вам могут не понадобиться все возможности, предлагаемые протоколом TLS. В этих случаях оптимальным решением станет *протокольный фреймворк Noise*.

Этот фреймворк устраняет временную сложность TLS, позволяя обойтись без переговоров в ходе рукопожатия. Клиент и сервер, применяющие Noise, следуют линейному протоколу, который не предусматривает разветвлений, в отличие от TLS, который может идти разными путями в зависимости от информации, содержащейся в различных сообщениях, передаваемых в ходе рукопожатия. Фреймворк Noise переносит всю сложность на этап проектирования.

Разработчики, планирующие использовать в своем приложении протокольный фреймворк Noise, должны выбрать его конкретную реализацию. (Именно поэтому он называется протокольным *фреймворком*, а не протоколом.) Таким образом, они должны решить, какие криптографические алгоритмы будут применяться, какая сторона соединения будет проходить аутентификацию, будут ли использоваться предварительно распределенные ключи и т. д. После этого протокол реализуется и превращается в жесткую серию сообщений, что впоследствии может стать проблемой, если вам потребуется обновить протокол, сохранив при этом обратную совместимость с устройствами, которые не могут быть обновлены.

9.5.1. Схемы рукопожатия фреймворка Noise

Протокольный фреймворк Noise предлагает на выбор множество *схем рукопожатия*, которые обычно имеют название, указывающее на то, что происходит в ходе их выполнения. Например, название схемы *IK* указывает на то, что открытый ключ клиента отправляется в процессе рукопожатия (первая буква *I* означает *immediate*, то есть «незамедлительно»), а открытый ключ сервера известен клиенту заранее (буква *K* означает *known*, то есть «известен»). После выбора схемы рукопожатия применяющие ее приложения уже никогда не будут пытаться задействовать другие схемы. С практической точки зрения это делает Noise более простым и линейным протоколом по сравнению с TLS.

В остальной части этого раздела для объяснения принципа работы фреймворка Noise я буду использовать схему рукопожатия под названием *NN*. Она довольно проста, но небезопасна из-за наличия в названии двух *N*, указывающих на то, что ни одна из сторон не проходит аутентификацию. На языке фреймворка Noise эта схема описывается так:

```
NN:
    -> e
    <-e, ee
```

Каждая строка представляет собой шаблон сообщения, а стрелка указывает направление его передачи. Каждый шаблон сообщения представляет собой последовательность токенов (здесь их всего два: `e` и `ee`), которые диктуют обеим сторонам соединения, что они должны делать.

- Последовательность `-> e` означает, что клиент должен сгенерировать пару эфемерных ключей и отправить открытый ключ серверу. Сервер интерпретирует это сообщение иначе: он должен получить эфемерный открытый ключ и сохранить его.
- Последовательность `<-e, ee` означает, что сервер должен сгенерировать пару эфемерных ключей и отправить открытый ключ клиенту, а затем обменяться с ним эфемерными ключами по протоколу Диффи — Хеллмана (DH). При этом эфемерный ключ клиента обозначается первой буквой `e`, а эфемерный ключ сервера — второй буквой `e`. В свою очередь, клиент должен получить от сервера эфемерный открытый ключ и использовать его для обмена ключами по протоколу DH.

ПРИМЕЧАНИЕ

Фреймворк Noise применяет комбинацию определенных токенов для указания различных типов рукопожатий. Например, токен *s* означает *static key* (статический или долгосрочный ключ) в противоположность эфемерному ключу, а токен *es* — что обмен ключами по протоколу DH должен выполняться с помощью эфемерного ключа клиента и статического ключа сервера.

Но это еще не все: в конце каждого шаблона сообщения (`-> e` и `<-e, ee`) отправитель передает также полезную нагрузку. Если до этого выполнялся обмен ключами по протоколу DH, чего не происходит в первом шаблоне сообщения `-> e`, полезная нагрузка шифруется и аутентифицируется. В конце рукопожатия оба участника формируют набор симметричных ключей и начинают шифровать сообщения, как в ходе применения протокола TLS.

9.5.2. Процесс рукопожатия при использовании фреймворка Noise

Одна из особенностей фреймворка Noise заключается в том, что он постоянно проверяет подлинность транскрипта рукопожатия. Для этого обе стороны хранят две переменные: хеш (`h`) и ключ цепочки (`ck`). Каждое отправленное или полученное сообщение рукопожатия хешируется предыдущим значением `h` (рис. 9.10).



Рис. 9.10. При использовании протокольного фреймворка Noise каждая сторона соединения отслеживает значения h всех сообщений, отправленных и полученных в ходе рукопожатия. Когда сообщение отправляется и шифруется с помощью алгоритма аутентифицированного шифрования с присоединенными данными (AEAD), текущее значение h задействуется в качестве присоединенных данных для аутентификации этапов рукопожатия, выполненных вплоть до этого момента

В конце каждого шаблона сообщения потенциально пустая полезная нагрузка шифруется с помощью алгоритма аутентифицированного шифрования с присоединенными данными (AEAD) (см. главу 4). Когда это происходит, значение h аутентифицируется по полю присоединенных данных AEAD. Это позволяет фреймворку Noise постоянно следить за тем, чтобы обе стороны соединения видели одну и ту же серию сообщений, следующих в одном и том же порядке.

Кроме того, каждый раз, когда происходит обмен ключами по протоколу DH (а в ходе рукопожатия таких эпизодов может быть несколько), его результат вместе с предыдущим ключом цепочки (ck) подается на вход функции HKDF, которая формирует новый ключ цепочки и новый набор симметричных ключей, предназначенные для аутентификации и шифрования последующих сообщений (рис. 9.11).

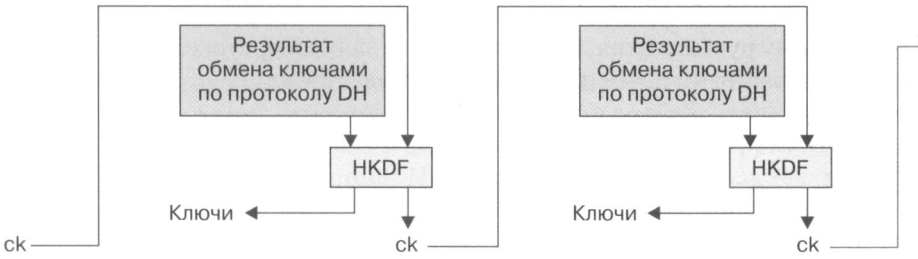


Рис. 9.11. При использовании протокольного фреймворка Noise каждая сторона соединения отслеживает ключ цепочки ck. Его значение применяется для формирования нового ключа цепочки и новых ключей шифрования, которые будут задействоваться при каждом обмене ключами по протоколу DH

Это делает Noise простым протоколом с точки зрения времени выполнения: в нем нет разветвлений и обе стороны соединения делают то, что должны. Библиотеки, реализующие фреймворк Noise, также чрезвычайно просты и содержат несколько сотен строк кода, в отличие от TLS-библиотек, содержащих сотни тысяч строк. Несмотря на то что фреймворк Noise более сложен в использовании и для его интеграции в приложение разработчикам требуется понимать принцип его работы, он является хорошей альтернативой протоколу TLS.

РЕЗЮМЕ

- Transport Layer Security (TLS) — это безопасный транспортный протокол, предназначенный для шифрования сообщений, которыми обмениваются машины. Ранее он назывался Secure Sockets Layer (SSL) и иногда до сих пор упоминается под этим названием.
- Протокол TLS работает поверх TCP и ежедневно используется для защиты соединений между браузерами, веб-серверами, мобильными приложениями и т. д.
- Для защиты UDP-соединений TLS предусматривает версию под названием Datagram Transport Layer Security (DTLS).
- TLS и большинство других безопасных транспортных протоколов предусматривают фазу рукопожатия, в ходе которой происходят безопасные переговоры, и фазу передачи данных, во время которой коммуникации шифруются с помощью ключей, полученных в первой фазе.
- Чтобы не задействовать веб-инфраструктуру открытых ключей, приложения, использующие протокол TLS, могут выполнить привязку сертификата и открытого ключа, разрешив устанавливать безопасные соединения только с помощью определенных сертификатов или открытых ключей.
- В качестве меры обеспечения глубоко эшелонированной защиты системы могут отзывать сертификаты (для удаления скомпрометированных) и выполнять их мониторинг (для выявления скомпрометированных сертификатов или ЦС).
- Чтобы устранить сложности, связанные с применением протокола TLS, при наличии контроля над обеими сторонами соединения можно использовать протокольный фреймворк Noise.
- Чтобы задействовать фреймворк Noise, при разработке протокола необходимо выбрать схему рукопожатия. Благодаря этому данный протокол гораздо более прост и безопасен, но менее гибок по сравнению с TLS.

10 Сквозное шифрование

В этой главе

- ✓ Сквозное шифрование и его важность.
- ✓ Различные попытки решения проблемы шифрования электронной почты.
- ✓ Как сквозное шифрование меняет системы обмена сообщениями.

В главе 9 мы говорили о таких безопасных транспортных протоколах, как TLS и Noise. Помимо этого, я потратил немало времени на объяснение того, что доверие в сети обеспечивается сотнями центров сертификации, которым доверяют ваш браузер и операционная система. Несмотря на свое несовершенство, эта система до сих пор оставалась вполне приемлемым решением для Всемирной паутины, которая представляет собой сложную сеть участников, ничего не знающих друг о друге.

Поиск способов обеспечения доверия другим людям (и их открытым ключам) и их масштабирования — центральная проблема реальной криптографии. При описании одной из областей исследований известный криптограф сказал: «Проблема симметричной криптографии решена». И по большей части это утверждение было верным. У нас редко возникают проблемы с шифрованием сообщений, и мы вполне уверены в алгоритмах шифрования, которые используем. Сегодня, когда речь заходит о шифровании, большинство инженерных задач оказываются связанными не с самими алгоритмами, а с личностями Алисы и Боба и способами их подтверждения.

Для установления доверия криптография предлагает не одно, а множество различных решений, практичность которых зависит от контекста. В этой главе я опишу некоторые из методов, с помощью которых люди и приложения устанавливают доверительные отношения между пользователями.

10.1. ЗАЧЕМ НУЖНО СКВОЗНОЕ ШИФРОВАНИЕ

Сквозное шифрование представляет собой не криптографический протокол, а концепцию защиты коммуникаций между двумя или более участниками от всевозможных действий противника. В начале книги я привел простой пример с королевой Алисой, которая хотела отправить сообщение лорду Бобу так, чтобы никто посторонний не смог его прочитать. Сегодня пользователи связываются друг с другом с помощью множества приложений, таких как электронная почта и системы обмена сообщениями, но большинство из них редко обеспечивают полноценное шифрование передаваемых посланий.

Вы можете спросить: а разве протокола TLS недостаточно? Теоретически, может, и достаточно. В главе 9 вы узнали о том, что TLS используется для защиты коммуникаций во многих сценариях. Однако сквозное шифрование — это концепция, учитывающая реальных людей. Напротив, протокол TLS чаще всего задействуется в системах, которые по своей сути предполагают наличие посредника (рис. 10.1). В таких системах TLS применяется только для защиты коммуникаций между центральным сервером и его пользователями, при этом сервер может просматривать весь трафик. По сути, эти серверы-посредники необходимы для работы приложения и являются *доверенными третьими сторонами* протокола. Иными словами, мы должны доверять этим компонентам системы, чтобы протокол считался безопасным (спойлер: такой протокол нельзя назвать достаточно хорошим).

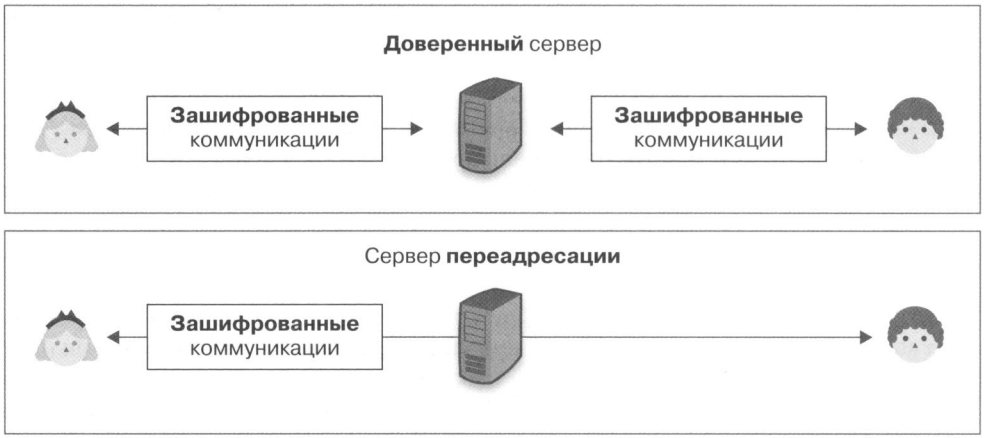


Рис. 10.1. В большинстве систем центральный сервер *(вверху)* передает сообщения от одного пользователя другому. Между пользователем и центральным сервером обычно устанавливается защищенное соединение, которое позволяет серверу просматривать все сообщения пользователя. Протокол, обеспечивающий сквозное шифрование *(внизу)*, шифрует сообщения на всем пути от отправителя до предполагаемого получателя, не позволяя ни одному серверу-посреднику просматривать сообщения в открытом виде

Однако существуют топологии и похуже. Обмен сообщениями между пользователем и сервером может проходить через множество транзитных участков, в качестве которых могут выступать машины, просматривающие трафик (их часто называют *промежуточными устройствами* (middleboxes)). Даже если этот трафик зашифрован, некоторые из таких *устройств* способны прерывать или *обрывать* TLS-соединения, а затем либо пересылать трафик в открытом виде, либо устанавливать другое TLS-соединение со следующим участком. Иногда TLS-соединение обрывается с благими намерениями, например для лучшей фильтрации трафика, балансировки соединений с учетом географии, в ходе работы в пределах центра обработки данных и т. д. Это увеличивает поверхность атаки, поскольку создает больше возможностей для просмотра сообщений. А иногда обрыв TLS-соединения является злонамеренным и используется для перехвата, записи и несанкционированного просмотра трафика.

В 2015 году компания Lenovo была поймана на продаже ноутбуков с предустановленным программным обеспечением и сертификатом (см. главу 9). Это ПО перехватывало HTTPS-трафик с помощью ЦС Lenovo и внедряло рекламу в веб-страницы. Более того, такие крупные страны, как Китай и Россия, были уличены в перенаправлении интернет-трафика через свои сети с целью просмотра и перехвата передаваемых сообщений. В 2013 году Эдвард Сноуден обнародовал огромное количество документов АНБ, доказывающих то, что многие правительства (не только США) шпионят за гражданами путем перехвата трафика, передаваемого по интернет-кабелям, связывающим мир воедино.

Владение пользовательскими данными и их просмотр также являются зоной ответственности компаний. Как я уже неоднократно упоминал, утечки и взломы происходят очень часто и могут иметь разрушительные последствия для репутации компании. С юридической точки зрения исполнение таких законов, как Общий регламент по защите данных (GDPR), могут стоить организациям больших денег. Правительственные запросы наподобие печально известных писем о национальной безопасности (National security letters, NSL), которые иногда не позволяют компаниям и заинтересованным лицам даже упоминать об их получении (это называется приказом о неразглашении), тоже могут быть источниками дополнительных издержек для них.

Таким образом, если вы пользуетесь каким-то популярным онлайн-приложением, есть вероятность, что одно или несколько правительств уже имеют или могут получить доступ ко всем опубликованным там сообщениям и загруженным файлам. В зависимости от *модели угроз* (она определяет то, от чего необходимо защититься) для приложения или его наиболее уязвимых пользователей сквозное шифрование может играть важную роль в обеспечении конфиденциальности и приватности конечных пользователей.

В этой главе мы рассмотрим различные техники и протоколы, которые были созданы для установления доверия между людьми. В частности, вы узнаете, как работает шифрование электронной почты и как безопасный обмен сообщениями меняет ландшафт сквозных зашифрованных коммуникаций.

10.2. КОРНЯ ДОВЕРИЯ НИГДЕ НЕ НАЙТИ

Рассмотрим один из простейших сценариев сквозного шифрования: Алиса хочет отправить Бобу зашифрованный файл через Интернет. Учитывая все те криптографические алгоритмы, о которых узнали в первых главах этой книги, вы наверняка представляете, как это можно осуществить. Например, так.

1. Боб отправляет Алисе свой открытый ключ.
2. Алиса шифрует файл открытым ключом Боба и отправляет его ему.

Возможно, Алиса и Боб могут встретиться вживую или использовать другой уже имеющийся у них безопасный канал, чтобы обменяться открытым ключом, содержащимся в первом сообщении. В этом случае мы говорим о том, что у них есть *внеполосный* (out-of-band) способ установления доверия. Однако так бывает не всегда. Представьте, что я написал свой открытый ключ в этой книге и попросил вас использовать его для отправки зашифрованного сообщения на мой адрес электронной почты. Что мешает редактору заменить мой открытый ключ своим?

То же самое и в случае с Алисой: как она может удостовериться в том, что полученный ею открытый ключ действительно принадлежит Бобу? Какой-нибудь посредник вполне мог подделать первое сообщение. Как будет показано в этой главе, криптография не дает четкого ответа на вопрос, связанный с доверием, она лишь предлагает различные решения, которые могут сработать в разных ситуациях. Причина отсутствия истинного решения заключается в том, что мы пытаемся объединить реальность (живых людей) с теоретическим криптографическим протоколом.

> Защита открытых ключей от взлома — самая сложная проблема при создании использующих их приложений. Это ахиллесова пята криптографии с открытым ключом, и в основном сложность программного обеспечения обусловлена попыткой решения именно этой проблемы.
>
> *Циммерманн и др. (PGP User's Guide Volume I: Essential Topics, 1992)*

Вернемся к рассмотрению сценария, в котором Алиса хочет отправить файл Бобу, и предположим, что у них есть лишь одно ненадежное соединение. В этом случае возникает неразрешимая проблема доверия. Алиса не может точно установить принадлежность открытого ключа Бобу. Это типичная проблема курицы и яйца. Тем не менее следует отметить, что если ни один *активный* посредник не подменит открытый ключ Боба, содержащийся в первом сообщении, то протокол окажется безопасным. Даже в случае пассивной записи сообщений злоумышленнику будет уже слишком поздно расшифровывать второе сообщение постфактум.

Разумеется, полагаться на то, что вероятность стать жертвой атаки посредника *не очень высока*, — не самая лучшая идея. К сожалению, часто у нас нет возможности этого избежать. Например, браузер Google Chrome поставляется с набором центров сертификации, которым он доверяет, но как вы получили сам Chrome? Возможно, с помощью использующегося по умолчанию в вашей операционной

системе браузера, который полагается на собственный набор центров сертификации. Но откуда он взялся? Он был установлен на купленном вами ноутбуке. А откуда взялся этот ноутбук? Как видите, «черепахи там до самого низа». В какой-то момент вам приходится поверить в то, что на более ранних этапах все было сделано правильно.

Типичная модель угроз учитывает ограниченное количество проблем, останавливаясь на какой-то определенной «черепахе». Поэтому в оставшейся части главы мы будем исходить из того, что у вас есть безопасный способ создания так называемого *корня доверия*. Все криптографические системы опираются на некий корень доверия, на основе которого протокол может выстроить защиту. Им может быть некое секретное или открытое значение, выступающее точкой отсчета для нашего протокола, или внеполосный канал, который мы можем использовать для его получения.

10.3. ПРОВАЛЫ В СФЕРЕ ШИФРОВАНИЯ ЭЛЕКТРОННОЙ ПОЧТЫ

Электронная почта создавалась как *незашифрованный* протокол и до сих пор таковым остается. В этом можно винить то время, когда о безопасности задумывались в последнюю очередь. Идея шифрования электронной почты начала реализовываться в 1991 году, после появления инструмента *Pretty Good Privacy* (PGP). Его создатель Фил Циммерманн решил выпустить PGP в ответ на появление законопроекта, который чуть было не стал законом ранее в том же году. Этот законопроект давал правительству США право получать все голосовые и текстовые сообщения от любой компании, работающей в сфере электронных коммуникаций. В конце своего эссе *Why Do You Need PGP?* (1994) Филипп Циммерманн пишет: «PGP позволяет людям взять обеспечение конфиденциальности в собственные руки. Общество испытывает возрастающую потребность в этом. Поэтому я его и создал».

Протокол *OpenPGP* был стандартизирован в 1998 году в документе RFC 2440 и получил широкое распространение после выпуска реализации *GNU Privacy Guard* (GPG) с открытым исходным кодом, который состоялся примерно в то же время. Сегодня GPG по-прежнему является основной реализацией, а термины GPG и PGP используются практически взаимозаменяемо.

10.3.1. PGP или GPG? И как все это работает

Протокол PGP, или OpenPGP, применяет гибридное шифрование (см. главу 6). Принцип его работы подробно описан в документе RFC 4880 (последней версии OpenPGP) и может быть сведен к следующему набору шагов.

1. Отправитель создает электронное письмо. На этом этапе его содержимое сжимается перед выполнением шифрования.
2. Реализация OpenPGP генерирует случайный симметричный ключ и с его помощью симметрично шифрует письмо.

3. Симметричный ключ асимметрично шифруется открытым ключом каждого адресата с помощью методов, описанных в главе 6.
4. Зашифрованные версии симметричных ключей всех адресатов объединяются с зашифрованным сообщением. Тело письма заменяется этим блоком данных и отправляется всем получателям.
5. Чтобы расшифровать электронное письмо, получатель использует свой закрытый ключ для расшифровки симметричного ключа, с помощью которого затем расшифровывает содержимое письма.

Учтите, что OpenPGP также определяет способ подписи электронного письма в целях аутентификации отправителя. Для этого открытый текст, составляющий тело письма, хешируется, а затем подписывается с помощью закрытого ключа отправителя. Далее подпись добавляется к сообщению перед его шифрованием на шаге 2. Наконец, чтобы получатель смог определить, какой открытый ключ ему следует использовать для проверки подписи, на шаге 4 открытый ключ отправителя отправляется вместе с зашифрованным письмом (рис. 10.2).

УПРАЖНЕНИЕ

Знаете ли вы, почему содержимое электронного письма сжимается перед шифрованием, а не после?

На первый взгляд в этой схеме нет ничего плохого. Кажется, что она не позволяет посредникам просмотреть содержимое вашего письма, хотя его тема и другие заголовки остаются незашифрованными.

ПРИМЕЧАНИЕ

Важно отметить, что криптографические системы не всегда способны скрыть все метаданные. В приложениях, предназначенных для обеспечения конфиденциальности, метаданные представляют серьезную проблему и в худшем случае позволяют деанонимизировать вас! Например, протоколы сквозного шифрования не позволяют расшифровать сообщения, которыми обмениваются пользователи, но могут помочь выяснить их IP-адреса, длину отправленных и полученных ими сообщений, установить, с кем они чаще всего общаются (их профили в социальной сети), и т. д. Сокрытие этих метаданных требует немалых усилий.

Однако, если присмотреться, недостатки PGP становятся очевидными. Стандарт OpenPGP и его основная реализация, GPG, используют старые алгоритмы, а улучшить ситуацию не позволяет обратная совместимость. Наиболее важной проблемой является то, что шифрование не подлежит аутентификации. Это значит, что любой человек, перехвативший неподписанное письмо, способен внести в его зашифрованное содержимое изменение, степень которого зависит от применяемого алгоритма шифрования. По одной лишь этой причине я бы никому не рекомендовал использовать PGP.

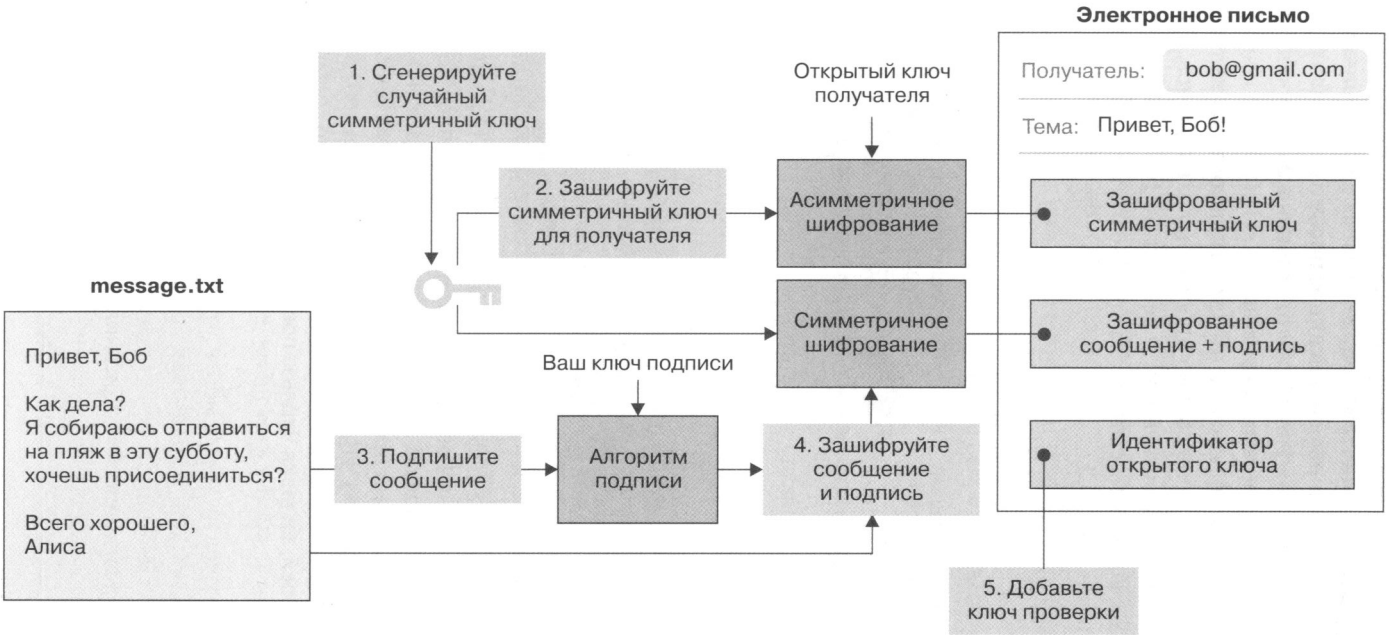


Рис. 10.2. Протокол PGP предназначен для шифрования и подписания сообщений. При интеграции с почтовыми клиентами он не скрывает тему и другие метаданные

Удивительный недостаток PGP заключается в том, что операции подписания и шифрования выполняются без должной осторожности. В 2001 году Дон Дэвис заметил, что такое наивное составление криптографических алгоритмов позволяет повторно зашифровать полученное подписанное письмо и отправить его другому получателю. Например, это позволяет Бобу отправить вам полученное от Алисы письмо так, как если бы его адресатом были вы!

Между тем подписание шифротекста вместо открытого текста тоже не лишено проблем, так как в этом случае можно просто удалить подпись, прилагаемую к шифротексту, заменив ее собственной. В результате Боб может притвориться отправителем письма, которое на самом деле пришло от Алисы. Эти две проблемы проиллюстрированы на рис. 10.3.



Рис. 10.3. В первом случае Алиса шифрует сообщение и подпись под ним с помощью открытого ключа Боба. Боб может повторно зашифровать это сообщение с помощью открытого ключа Чарльза, заставив того поверить, будто оно изначально предназначалось ему. Так работает протокол PGP. Во втором случае Алиса шифрует сообщение, предназначенное для Чарльза. При этом вместо открытого текста она подписывает зашифрованное сообщение. Боб, перехвативший это сообщение, может заменить ее подпись своей, заставив Чарльза поверить в то, что автором сообщения является он

УПРАЖНЕНИЕ

Можете ли вы придумать способ подписания сообщения, лишенный подобной двусмысленности?

Вишенкой на торте является то, что данный алгоритм не обеспечивает *прямую секретность* по умолчанию. А без нее компрометация вашего закрытого ключа позволяет расшифровать все отправленные вам ранее письма, зашифрованные с его помощью. Конечно, вы можете обеспечить прямую секретность, изменив свой ключ PGP, но этот процесс не так прост (например, вы можете подписать свой новый ключ старым), и большинство пользователей не утруждают себя этим. В заключение напомню следующее.

- Протокол PGP применяет старые криптографические алгоритмы.
- PGP не предусматривает аутентифицированное шифрование, поэтому небезопасен при использовании без подписей.
- Из-за плохого дизайна при получении подписанного сообщения вы не можете быть уверены в том, что являетесь его адресатом.
- Данный протокол не обеспечивает прямую секретность по умолчанию.

10.3.2. Масштабирование доверия между пользователями с помощью сети доверия

В PGP есть еще одна интересная особенность, связанная с тем, как вы получаете открытые ключи других людей и как понимаете, что им можно доверять. Дело в том, что при использовании этого протокола вы определяете степень доверия самостоятельно!

Что это значит? Представьте, что вы установили программу GPG и решили зашифровать несколько сообщений, адресованных друзьям. Для начала нужно найти безопасный способ получения их открытых PGP-ключей. Один из самых надежных способов сделать это — личная встреча, в ходе которой вы можете записать их открытые ключи на листе бумаги, а вернувшись домой, ввести их в программу на своем ноутбуке. После этого вы сможете отправлять друзьям сообщения, подписанные и зашифрованные с помощью OpenPGP. Но это довольно утомительно. Обязательно ли проделывать все это с каждым адресатом? Конечно, нет. Рассмотрим следующий сценарий.

- Вы получили открытый ключ Боба при личной встрече и, следовательно, доверяете ему.
- У вас нет открытого ключа Марка, но он есть у Боба, и тот ему доверяет.

Подумайте о том, что нужно сделать, чтобы вы могли доверять открытому ключу Марка. Боб может просто подписать ключ Марка, продемонстрировав вам свою уверенность в существовании связи между этим открытым ключом и электронной почтой Марка. Если вы доверяете Бобу, то можете доверять и открытому ключу Марка. В этом заключается суть концепции *децентрализованного* доверия PGP, которая называется *сетью доверия* (web of trust, WOT) (рис. 10.4).

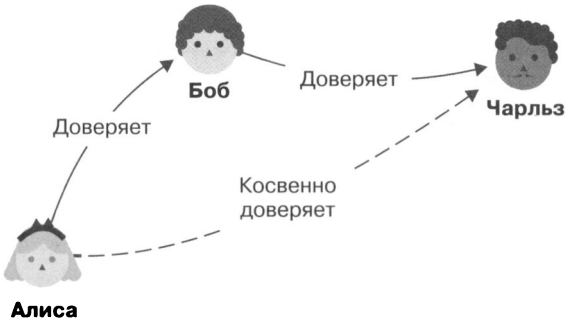


Рис. 10.4. Сеть доверия — это концепция, согласно которой пользователи могут транзитивно доверять другим пользователям, полагаясь на подписи. На схеме мы видим, что Алиса доверяет Бобу, который доверяет Чарльзу. Чтобы тоже доверять Чарльзу, Алиса может применить подпись Боба под его открытым ключом

Иногда на конференциях проводятся специальные вечеринки, участники которых, встретившись вживую, подписывают свои открытые ключи. Но по большей части это игра, так как на практике мало кто использует WOT для расширения своего PGP-круга.

10.3.3. Обнаружение ключей — это настоящая проблема

Протокол PGP предложил еще один способ решения проблемы обнаружения открытых ключей — *реестры ключей*. Данная концепция довольно проста и сводится к публикации вашего открытого PGP-ключа и связанных с ним подписей других людей, подтверждающих вашу личность, в некоем общедоступном списке. На практике это не работает, поскольку любой человек может опубликовать ключ и связанную с ним подпись, якобы соответствующие вашему адресу электронной почты. Более того, некоторые злоумышленники намеренно подделывают ключи, хранящиеся на серверах ключей, правда, скорее всего, они делают это для того, чтобы посеять хаос, а не для перехвата электронных писем. В некоторых случаях мы можем смягчить модель угроз и позволить доверенному органу подтверждать личности пользователей и их открытые ключи. Например, подумайте о компании, управляющей электронной почтой своих сотрудников.

В 1995 году компания RSA предложила стандарт *Secure/Multipurpose Internet Mail Extensions* (S/MIME) в качестве расширения для формата MIME (который, в свою очередь, является расширением для стандарта электронной почты) и альтернативы протоколу PGP. Стандарт S/MIME, описанный в документе RFC 5751, отличается от WOT тем, что использует для установления доверия инфраструктуру открытых ключей. В этом заключается практически единственное концептуальное отличие S/MIME от PGP. Поскольку в компаниях существуют процессы адаптации сотрудников, им имеет смысл задействовать такие протоколы, как S/MIME, для укрепления доверия во внутренней почтовой экосистеме.

Важно отметить, что PGP и S/MIME обычно используются поверх протокола *Simple Mail Transfer Protocol* (SMTP), который сейчас применяется для отправки и получения электронных писем. PGP и S/MIME были изобретены позднее, поэтому их интеграция с SMTP и почтовыми клиентами далека от совершенства. Например, шифруется только тело письма, а тема и другие заголовки остаются незашифрованными. S/MIME, как и PGP, — довольно старый протокол, использующий устаревшие криптографические алгоритмы и методы. Как и PGP, он не предусматривает аутентифицированного шифрования.

Недавнее исследование (Efail: *Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels*), посвященное интеграции обоих протоколов в почтовые клиенты, показало, что большинство их уязвимы к *эксфильтрации данных* — виду атак, позволяющему злоумышленнику получить доступ к содержимому зашифрованных писем путем отправки их поддельных версий адресатам.

В конце концов, эти недостатки могут и не иметь значения, поскольку большинство электронных писем, отправляемых и получаемых пользователями по всему миру, перемещаются по глобальной сети без шифрования. PGP показался довольно сложным в применении как далеким от технологий, так и продвинутым пользователям, которым приходится разбираться во многих тонкостях, чтобы зашифровать свою электронную почту. Например, нередко пользователи отвечают на зашифрованное письмо без помощи шифрования, цитируя все его содержимое в виде открытого текста. Ситуацию усугубляет еще и слабая или вообще отсутствующая поддержка PGP популярными почтовыми клиентами.

> В 1990-х годах я с восторгом смотрел в будущее, мечтая о мире, в котором все смогут установить GPG. Сейчас я по-прежнему с восторгом смотрю в будущее, но мечтаю о мире, в котором GPG можно удалить.
>
> *Мокси Марлинспайк (GPG and Me, 2015)*

По этим причинам PGP постепенно утрачивает поддержку (например, стандартная библиотека Golang перестала поддерживать PGP в 2019 году), в то время как все большее число криптографических приложений стремится заменить PGP и решить проблемы, связанные с удобством его использования. Сегодня трудно утверждать, что шифрование электронной почты когда-либо будет иметь такой же успех и широкое распространение, как, например, протокол HTTPS.

> Если сообщения могут быть отправлены в виде открытого текста, они будут отправлены именно в таком виде. По умолчанию электронная почта не предполагает сквозного шифрования. Основой электронной почты является открытый текст. Все популярные почтовые программы предполагают работу с открытым текстом. По большому счету, существующая в Интернете система электронной почты просто не предназначена для шифрования.
>
> *Томас Птачек (Stop Using Encrypted Email, 2020)*

10.3.4. Если не PGP, то что?

На нескольких предыдущих страницах я рассказал о множестве проблем, которые могут возникнуть при практическом применении такого простого механизма, как PGP. Да, я бы не рекомендовал его использовать. Хотя шифрование электронной почты все еще остается нерешенной проблемой, в настоящее время разрабатываются альтернативы PGP, которые можно задействовать в различных сценариях.

Одной из таких альтернатив является протокол и формат сообщений *saltpack*, призванный исправить некоторые из описанных ранее недостатков PGP. В 2021 году основными реализациями saltpack были keybase (https://keybase.io) и keys.pub (https://keys.pub). Интерфейс инструмента keys.pub показан на рис. 10.5.

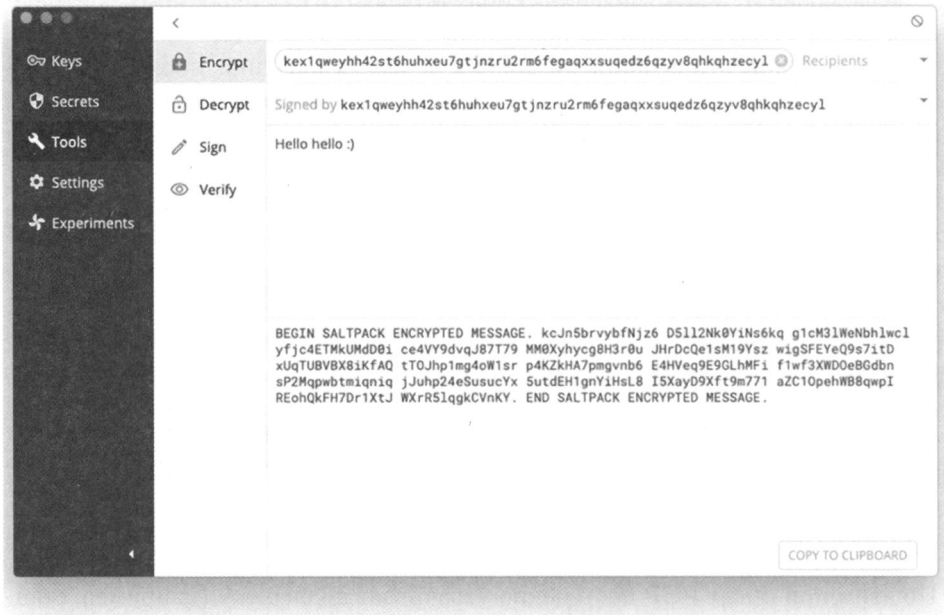


Рис. 10.5. keys.pub — это приложение для настольных компьютеров, реализующее протокол saltpack. С его помощью вы можете импортировать открытые ключи других людей, а также шифровать и подписывать адресованные им сообщения

Все эти реализации отходят от WOT и позволяют пользователям публиковать свои открытые ключи в различных социальных сетях, чтобы связать с ними свою личность (рис. 10.6). Очевидно, что протокол PGP не мог предусмотреть этот механизм обнаружения ключей, поскольку появился еще до расцвета социальных сетей.

Но защищенные коммуникации в наше время не ограничиваются передачей одноразовых сообщений, так что применение этих инструментов становится все менее актуальным. В следующем разделе мы поговорим о *безопасном обмене сообщениями* — об одной из областей, призванных заменить коммуникационный аспект протокола PGP.

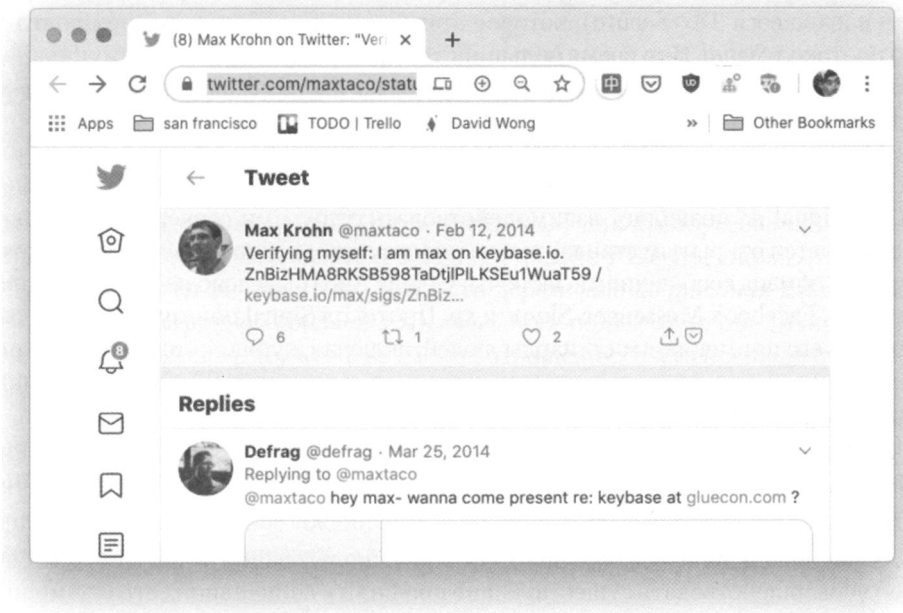


Рис. 10.6. Пользователь keybase публикует свой открытый ключ в социальной сети Twitter. Это позволяет другим пользователям получить дополнительное подтверждение связи его личности с определенным открытым ключом

10.4. БЕЗОПАСНЫЙ ОБМЕН СООБЩЕНИЯМИ. СОВРЕМЕННЫЙ ВЗГЛЯД НА СКВОЗНОЕ ШИФРОВАНИЕ НА ПРИМЕРЕ ПРОТОКОЛА SIGNAL

В 2004 году в техническом документе *Off-the-Record Communication, or, Why Not To Use PGP* был представлен протокол *Off-The-Record* (OTR). В отличие от PGP и S/MIME, OTR используется для шифрования не электронных писем, а сообщений в чате и является расширением протокола XMPP (Extensible Messaging and Presence Protocol — расширяемый протокол обмена сообщениями и информацией о присутствии).

Одна из отличительных особенностей OTR — *отрицаемость*, которая не позволяет получателям ваших сообщений и пассивным наблюдателям использовать отправленные вами сообщения в суде. Поскольку отправляемые вами сообщения аутентифицируются и шифруются симметрично с помощью ключа, который имеется как у вас, так и у получателя, они с легкостью могут быть подделаны получателями. Напротив, при использовании протокола PGP сообщения подписываются, что делает авторство *неопровержимым*. Насколько мне известно, ни одно из этих свойств еще не становилось предметом судебных разбирательств.

В 2010 году было выпущено приложение для мобильных телефонов Signal (тогда именовавшееся TextSecure), которое применяло разработанный незадолго до этого протокол *Signal*. В то время большинство протоколов защиты коммуникаций, таких как PGP, S/MIME и OTR, основывались на так называемых *федеративных протоколах*, которые не требовали центрального органа для обеспечения работы сети. Мобильное приложение Signal в значительной степени отошло от традиций, предложив центральную службу и единое официальное клиентское приложение.

Хотя Signal не позволяет взаимодействовать с другими серверами, протокол Signal является открытым стандартом и используется многими другими приложениями для обмена сообщениями, включая Google Allo (выведено из эксплуатации), WhatsApp, Facebook Messenger, Skype и др. Протокол Signal пользуется огромным успехом, и его применяют миллиарды людей, включая журналистов, людей, скрывающихся от слежки правительственных органов, и даже мою 92-летнюю бабушку (клянусь, я не заставлял ее его устанавливать).

Протокол Signal призван исправить многие из недостатков PGP, о которых я говорил ранее. В этом разделе расскажу о каждой из следующих интересных особенностей этого протокола.

- Как мы можем добиться большего, чем при использовании WOT? Есть ли способ усовершенствовать существующие профили в социальной сети с помощью сквозного шифрования? Ответ Signal заключается в применении такого подхода, как *доверие при первом использовании* (trust on first use, TOFU). TOFU позволяет пользователям проявить доверие к другим пользователям при первом контакте для создания долговременного безопасного канала связи. Затем пользователи могут в любой момент проверить, не подвергся ли их первый обмен сообщениями атаке посредника, сопоставив свои сеансовые секреты с помощью какого-то вспомогательного канала связи.
- Как мы можем усовершенствовать протокол PGP, чтобы обеспечить прямую секретность каждый раз, когда начинаем разговор с кем-либо? Первая часть протокола Signal, как и в случае с большинством безопасных транспортных протоколов, предполагает обмен ключами, но в данном случае для этого используется особый протокол *Extended Triple Diffie-Hellman* (X3DH). Подробнее об этом — позже.
- Как мы можем усовершенствовать PGP, чтобы обеспечить прямую секретность для каждого сообщения? Это важно, потому что общение между пользователями может длиться годами и его компрометация в определенный момент не должна приводить к раскрытию всей многолетней переписки. Протокол Signal решает эту задачу с помощью так называемого *симметричного храповика*.
- Что, если в какой-то момент сеансовые секреты двух пользователей будут скомпрометированы? Означает ли это конец игры? Можем ли мы восстановиться после этого? Протокол Signal вводит новое свойство безопасности, называемое *безопасностью после компрометации* (post-compromise security, PCS), которая обеспечивается с помощью так называемого *храповика Диффи — Хеллмана* (Diffie-Hellman ratchet).

Итак, приступим! Для начала разберемся, как работает подход TOFU в рамках протокола Signal.

10.4.1. Большее удобство для пользователя по сравнению с WOT: доверяй, но проверяй

Один из главных провалов в сфере шифрования электронной почты был обусловлен применением протокола PGP и модели WOT для преобразования профилей в социальной сети в *безопасные*. Исходный дизайн PGP предполагал, что люди будут встречаться лично, чтобы провести *церемонию подписания ключей*, также называемую *вечеринкой подписания ключей*, для подтверждения личностей друг друга, но это оказалось неудобно во многих отношениях. Сегодня люди очень редко подписывают PGP-ключи друг друга.

При использовании таких приложений, как PGP, OTR, Signal и пр., большинство людей слепо доверяют ключу при первом же взаимодействии и отвергают любые последующие изменения (рис. 10.7). Таким образом, атаке посредника может подвергнуться только первое соединение, и то лишь при условии активных действий со стороны злоумышленника.

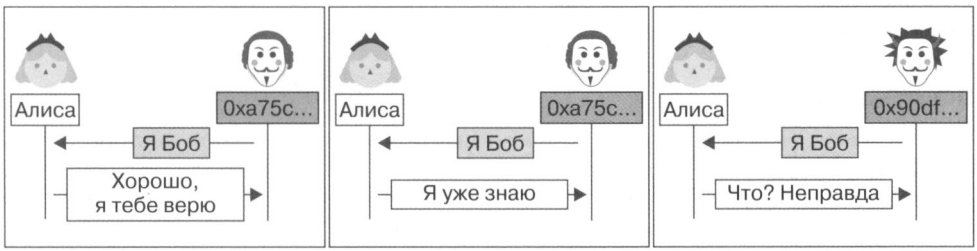


Рис. 10.7. Доверие при первом использовании позволяет Алисе доверять первому соединению и не доверять последующим, если для их установления не применяется тот же самый открытый ключ. TOFU — это простой механизм для установления доверия, хорошо подходящий для случаев, когда вероятность того, что первое соединение будет активно атаковано посредником, невелика. Связь между открытым ключом и личностью (в данном случае Боба) также может быть проверена постфактум с помощью другого канала

Хотя TOFU не идеальная модель безопасности, зачастую это лучшее, что у нас есть, и она уже доказала свою чрезвычайную полезность. Например, использование протокола Secure Shell (SSH) часто предполагает доверие открытому ключу сервера во время первого соединения (рис. 10.8) и неприятие любых изменений в дальнейшем.

Несмотря на то что TOFU-системы доверяют впервые увиденному ключу, они позволяют пользователю впоследствии убедиться в правильности этого ключа и выявить возможные попытки злоумышленников выдать себя за другое лицо. Пользователи реальных приложений обычно сравнивают так называемые

отпечатки пальцев, которые чаще всего представляют собой шестнадцатеричные представления открытых ключей или их хеши. Эта проверка, разумеется, выполняется с помощью внеполосного канала, поскольку в случае компрометации SSH-соединения сама проверка тоже будет скомпрометирована.

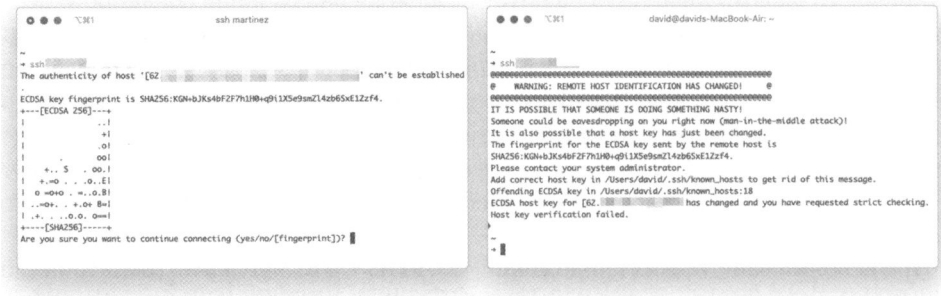


Рис. 10.8. SSH-клиенты применяют подход TOFU. При первом подключении к SSH-серверу *(слева)* вы можете слепо поверить в наличие связи между SSH-сервером и отображаемым открытым ключом. Если впоследствии открытый ключ SSH-сервера изменится *(справа)*, ваш SSH-клиент не позволит вам подключиться к нему

ПРИМЕЧАНИЕ

Разумеется, если пользователи не проверяют отпечатки пальцев, то они могут стать жертвами атаки посредника, даже не подозревая об этом. Однако это компромисс, на который приходится идти для масштабного внедрения сквозного шифрования. Действительно, провал WOT отчетливо продемонстрировал, что широкое распространение приложений, ориентированных на обеспечение безопасности, во многом зависит от удобства их применения.

В мобильном приложении Signal отпечаток пальца для соединения Алисы и Боба вычисляется путем:

- хеширования идентификационного ключа Алисы с префиксом в виде ее имени пользователя (в Signal это номер телефона) и интерпретации усечения этого дайджеста в качестве серии чисел;
- выполнения того же действия для Боба;
- отображения результата конкатенации этих двух серий чисел на экране пользователя.

Приложения, подобные Signal, задействуют *QR-коды*, чтобы облегчить пользователям проверку отпечатков пальцев, поскольку эти коды могут быть довольно длинными (рис. 10.9).

Теперь рассмотрим принцип работы протокола Signal, в частности то, как он обеспечивает прямую секретность.



Рис. 10.9. Работая с приложением Signal, вы можете проверить подлинность и конфиденциальность соединения с помощью другого канала связи (как в реальной жизни), чтобы убедиться в том, что ваши с другом отпечатки пальцев (в приложении Signal они называются кодами безопасности) совпадают. Проще всего это можно сделать с помощью QR-кода, который представляет эту информацию в удобном для сканирования формате. Приложение Signal также хеширует сеансовый секрет вместо открытых ключей двух пользователей, что позволяет им проверять одну большую строку вместо двух

10.4.2. X3DH — рукопожатие в рамках протокола Signal

До появления Signal большинство приложений для безопасного обмена сообщениями были *синхронными*. Это означает, что, например, Алиса не могла начать (или продолжить) защищенный сквозным шифрованием разговор с Бобом при его отсутствии в сети. Протокол Signal *асинхронный* (как электронная почта), что позволяет Алисе начать (и продолжить) разговор с пользователями, которые находятся вне сети.

Как вы помните, *прямая секретность* (см. главу 9) означает, что компрометация ключей не приводит к компрометации предыдущих сеансов и что обмен ключами является интерактивным, поскольку обе стороны должны генерировать эфемерные ключи по протоколу Диффи — Хеллмана (DH). Далее вы узнаете, как протокол

Signal реализует *неинтерактивный* обмен ключами (в это время одна из сторон может находиться вне сети), обеспечивая прямую секретность. Итак, приступим.

Чтобы начать общение с Бобом, Алиса инициирует процесс обмена ключами с ним. Протокол Signal объединяет три и более эпизода обмена DH-ключами в один (X3DH). Но прежде, чем вы узнаете, как это работает, вам следует познакомиться с тремя типами DH-ключей, которые применяет Signal.

- *Ключи идентификации.* Это долгосрочные ключи, которые представляют пользователей. Как вы понимаете, если бы протокол Signal задействовал только идентификационные ключи, то его принцип работы был бы очень похож на PGP и он не обеспечивал бы никакой прямой секретности.
- *Одноразовые открытые ключи.* Для обеспечения прямой секретности в ходе обмена ключами даже в том случае, когда одна из сторон находится вне сети, протокол Signal требует, чтобы пользователи загрузили несколько *одноразовых* открытых ключей, представляющих собой простые эфемерные ключи, которые загружаются заранее и удаляются после применения.
- *Подписанные открытые ключи.* На этом можно было бы и остановиться, однако описанный ранее принцип работы не учитывает один крайний случай. Поскольку одноразовые ключи, загружаемые пользователем, в какой-то момент могут закончиться, он должен загружать также подписанный *среднесрочный* открытый ключ. Таким образом, даже если на сервере больше нет одноразовых ключей, связанных с вашим именем пользователя, кто-то все равно сможет задействовать подписанный вами ключ, чтобы обеспечить прямую секретность вплоть до момента последнего изменения данного ключа. Это также означает, что вам придется периодически менять подписанный открытый ключ (например, раз в неделю).

Общая схема процесса создания беседы в Signal представлена на рис. 10.10.

Рассмотрим каждый из этих этапов более подробно. Сначала пользователь регистрируется, отправляя при этом:

- один ключ идентификации;
- один подписанный открытый ключ и его подпись;
- определенное количество одноразовых открытых ключей.

После этого его обязанностью становится периодическая смена подписанного открытого ключа и загрузка новых одноразовых открытых ключей (рис. 10.11).

ПРИМЕЧАНИЕ

Протокол Signal предполагает использование ключа идентификации для выполнения подписей при создании подписанных открытых ключей и в ходе обмена ключами по протоколу X3DH. Хотя я не рекомендую применять один и тот же ключ для разных целей, разработчики протокола Signal провели анализ и пришли к выводу, что в данном случае проблем быть не должно. Это не означает, что такой подход сработает в *вашем* случае и с *вашим* алгоритмом обмена ключами. Но я бы не советовал задействовать один ключ для разных целей.



Рис. 10.10. Рабочий процесс протокола Signal начинается с того, что пользователь регистрируется с помощью нескольких открытых ключей. Если Алиса хочет поговорить с Бобом, она сначала получает набор его открытых ключей, затем обменивается этими ключами по протоколу X3DH и создает исходное сообщение, используя результат обмена ключами. После получения сообщения Боб может выполнить аналогичные действия, чтобы инициировать и продолжить разговор

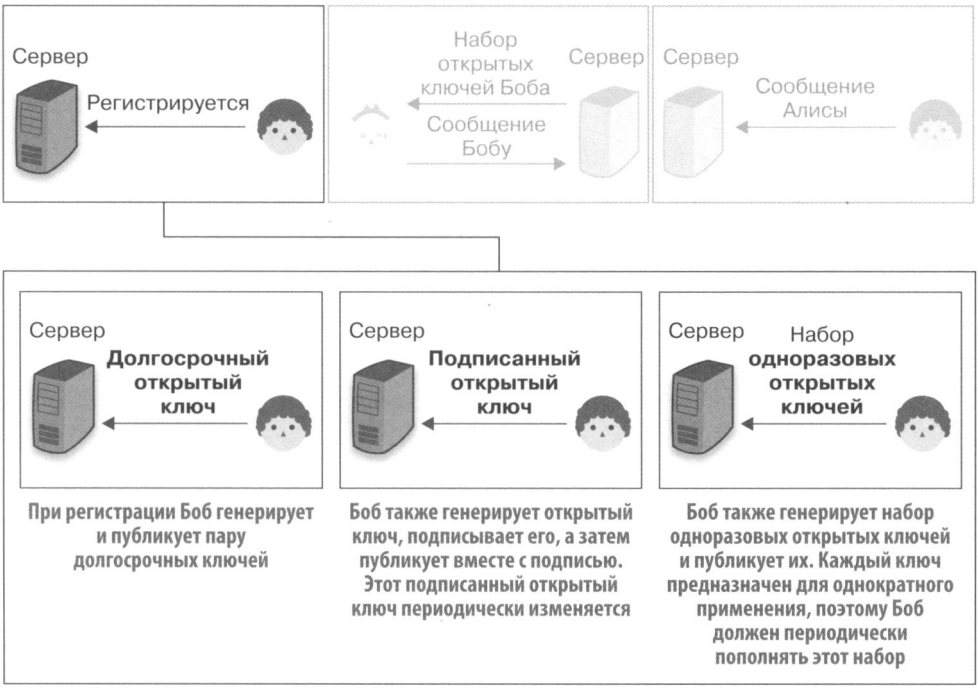


Рис. 10.11. Дополнение к схеме, приведенной на рис. 10.10. На первом этапе пользователь должен зарегистрироваться, сгенерировав несколько пар DH-ключей и отправив их открытые части на центральный сервер

Вернемся к примеру. После шага, представленного на рис. 10.11, Алиса начинает разговор с Бобом, запрашивая:

- ключ идентификации Боба;
- текущий подписанный открытый ключ Боба и связанную с ним подпись;
- если таковые еще имеются, то один из одноразовых открытых ключей Боба (после этого сервер удаляет одноразовый ключ, отправленный Алисе).

Алиса проверят подлинность подписи, связанной с подписанным открытым ключом, а затем выполняет X3DH-рукопожатие с помощью:

- всех открытых ключей Боба;
- пары эфемерных ключей, которую она генерирует специально для этого случая, чтобы обеспечить прямую секретность;
- ее собственного ключа идентификации.

Результат X3DH-рукопожатия используется в рамках протокола, который применяется после X3DH для шифрования сообщений Алисы, адресованных Бобу (подробнее об этом — в следующем разделе). X3DH-рукопожатие состоит из трех (опционально из четырех) эпизодов обмена DH-ключами, сгруппированных в один. Обмен DH-ключами выполняется между:

- ключом идентификации Алисы и подписанным открытым ключом Боба;
- эфемерным ключом Алисы и ключом идентификации Боба;
- эфемерным ключом Алисы и подписанным открытым ключом Боба;
- если у Боба остался одноразовый открытый ключ, то между этим ключом и эфемерным ключом Алисы.

Для получения выходного значения X3DH-рукопожатия результаты всех обменов DH-ключами конкатенируются и передаются функции формирования ключей (KDF), о которой мы говорили в главе 8. Разные эпизоды обмена ключами обеспечивают разные свойства. Первый и второй нужны для взаимной аутентификации, а последние два — для обеспечения прямой секретности. Все это довольно хорошо описано в спецификации протокола X3DH (https://signal.org/docs/specifications/x3dh/), которую я рекомендую прочитать, если вы хотите узнать больше. Описанный процесс схематично представлен на рис. 10.12.

Теперь Алиса может отправить Бобу свой открытый ключ идентификации, эфемерный открытый ключ, который она сгенерировала для начала разговора, и другую актуальную информацию, например, о том, какой из одноразовых открытых ключей Боба применила. Получив сообщение, Боб может использовать содержащиеся в нем открытые ключи, чтобы выполнить точно такой же обмен по протоколу X3DH. (По этой причине я не включил в иллюстрацию последний этап этого процесса.) Если Алиса задействовала один из одноразовых открытых ключей Боба, тот избавляется от него. Далее мы поговорим о том, что происходит после завершения X3DH-рукопожатия.

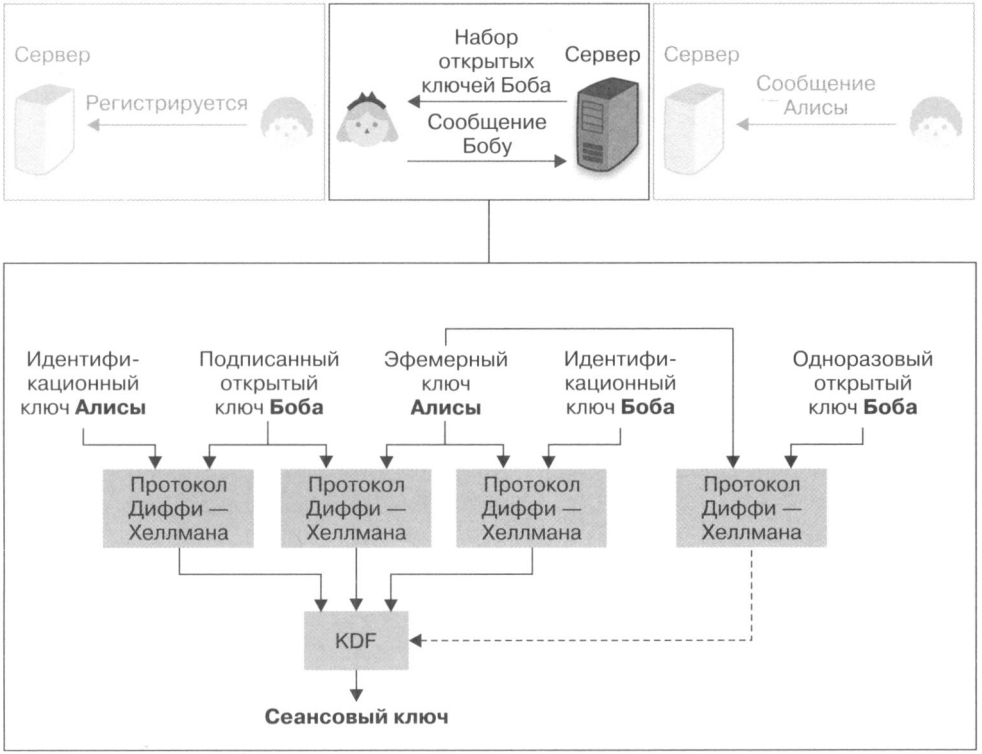


Рис. 10.12. Дополнение к схеме, приведенной на рис. 10.10. Чтобы отправить сообщение Бобу, Алиса получает набор его открытых ключей, содержащий долгосрочный ключ Боба, его подписанный открытый ключ и опционально один из его одноразовых открытых ключей. После нескольких эпизодов обмена различными ключами их результаты конкатенируются и подаются на вход функции формирования ключа для получения выходного значения, используемого в рамках следующего протокола для шифрования сообщений, адресованных Бобу

10.4.3. Двойной храповик — протокол Signal, используемый после рукопожатия

Фаза, следующая за X3DH-рукопожатием, длится до тех пор, пока пользователи не удалят свои разговоры или не потеряют какой-то из ключей. По этой причине, а также потому, что протокол Signal разрабатывался с учетом SMS-переписки, где временной интервал между двумя сообщениями может измеряться месяцами, данный протокол обеспечивает *прямую секретность* на уровне отдельных сообщений. В этом разделе вы узнаете о том, как работает алгоритм *двойного храповика* (double ratchet), используемый после выполнения рукопожатия.

Но сначала представьте себе простой протокол, применяемый после X3DH-рукопожатия. Алиса и Боб могли бы задействовать результат X3DH-рукопожатия

в качестве сеансового ключа для шифрования отправляемых друг другу сообщений (рис. 10.13).



Рис. 10.13. Наивный подход предполагает применение результата X3DH-рукопожатия в качестве сеансового ключа для шифрования сообщений, которыми обмениваются Алиса и Боб

Однако мы, как правило, стараемся применять разные ключи для разных целей. Что можно сделать, так это использовать результат X3DH-рукопожатия в качестве *зерна* (или *корневого ключа* согласно спецификации алгоритма двойного храповика) для функции формирования ключей (KDF), чтобы получить два других ключа. Алиса может использовать один ключ для шифрования сообщений, адресованных Бобу, а Боб — другой ключ для шифрования сообщений, адресованных Алисе (рис. 10.14).

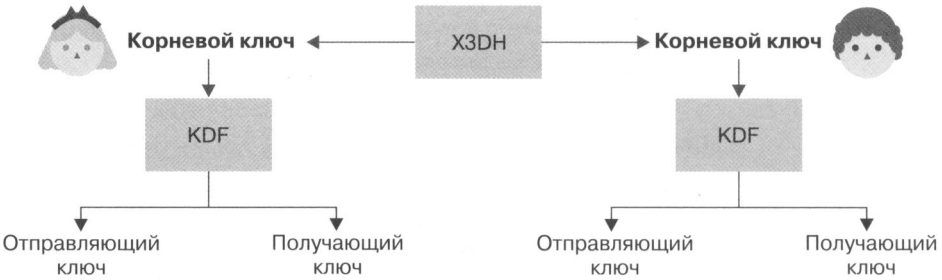


Рис. 10.14. Дополнение к схеме, приведенной на рис. 10.13. Более совершенный протокол, использующийся после X3DH-рукопожатия, предполагает применение функции формирования ключей к результатам обмена ключами с целью создания разных ключей для шифрования сообщений Боба и Алисы. В данном случае отправляющий ключ Алисы совпадает с получающим ключом Боба, а отправляющий ключ Боба — с получающим ключом Алисы

Этого могло бы быть достаточно, однако протокол Signal предполагает, что сеансы обмена текстовыми сообщениями могут длиться годами, в отличие от краткосрочных сеансов TLS, о которых мы говорили в главе 9. Поэтому, если в какой-то момент сеансовый ключ будет украден, все ранее записанные сообщения могут быть расшифрованы!

Чтобы этого не допустить, протокол Signal задействует так называемый *симметричный храповик* (рис. 10.15). Теперь вместо *отправляющего ключа* используется *отправляющий цепной ключ*, который не применяется непосредственно для шифрования сообщений. При отправке сообщений Алиса постоянно подает отправляющий цепной ключ на вход односторонней функции, которая формирует очередной отправляющий цепной ключ, а также фактические ключи для

шифрования отправляемых ею сообщений. Бобу приходится делать то же самое, но только с получающим цепным ключом. Таким образом, скомпрометировав один отправляющий ключ или отправляющий цепной ключ, злоумышленник не сможет восстановить предыдущие ключи. То же самое касается получаемых сообщений.

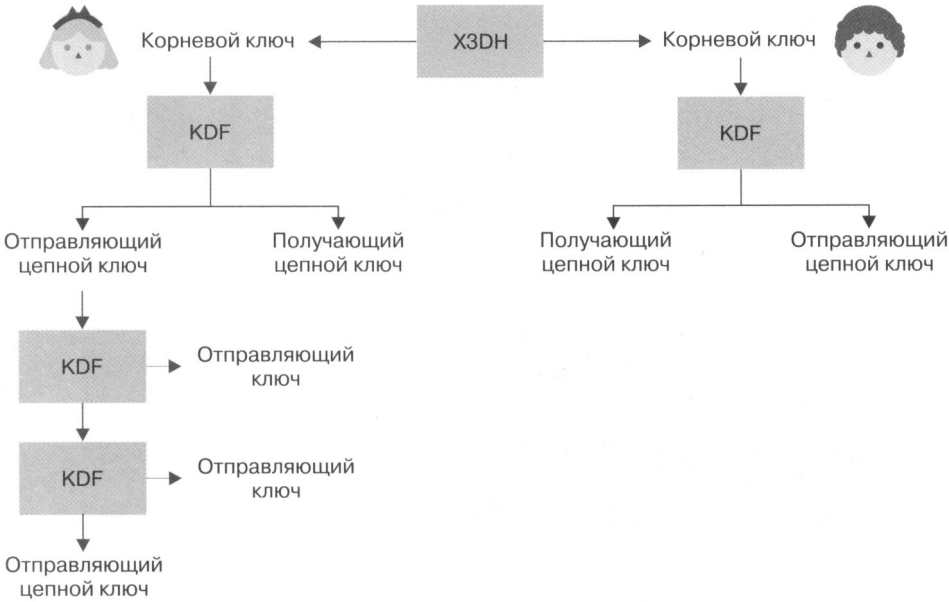


Рис. 10.15. Дополнение к схеме, приведенной на рис. 10.14. Для обеспечения прямой секретности после выполнения X3DH-рукопожатия мы можем подавать на вход функции KDF цепные ключи при отправке и получении каждого сообщения. Таким образом, компрометация отправляющего или получающего цепного ключа не позволяет злоумышленнику восстановить предыдущие

Итак, теперь прямая секретность обеспечивается в протоколе на уровне отдельных сообщений. Каждое отправленное и полученное сообщение защищает все те сообщения, которые были отправлены и получены ранее. Правда, данное утверждение весьма спорно, так как злоумышленник, скомпрометировавший ключ, скорее всего, сделает это путем компрометации телефона пользователя, который наряду с ключом, вероятно, будет содержать все предыдущие сообщения в открытом виде. Тем не менее, если оба участника разговора решат удалить предыдущие сообщения, например воспользовавшись функцией исчезающих сообщений в приложении Signal, прямая секретность все равно будет обеспечена.

Еще одной интересной особенностью протокола Signal является *безопасность после компрометации* (post-compromise security, PCS), также называемая *обратной секретностью*, о которой мы говорили в главе 8. Суть данной идеи заключается в возможности восстановления в случае компрометации ключей благодаря способности протокола к самоисцелению. Разумеется, если после компрометации у злоумышленника все еще будет доступ к вашему устройству, это мало что вам даст.

Безопасность после компрометации может быть обеспечена только путем повторного привнесения энтропии, к которой у злоумышленника, реализующего атаки не на постоянной основе, не было бы доступа. Эта энтропия должна быть одинаковой для обоих участников. Для ее обнаружения протокол Signal непрерывно выполняет обмен ключами с помощью механизма, называемого *DH-храповиком* или *храповиком Диффи — Хеллмана*. Каждое сообщение, отправленное протоколом, содержит текущий открытый храповый ключ (рис. 10.16).

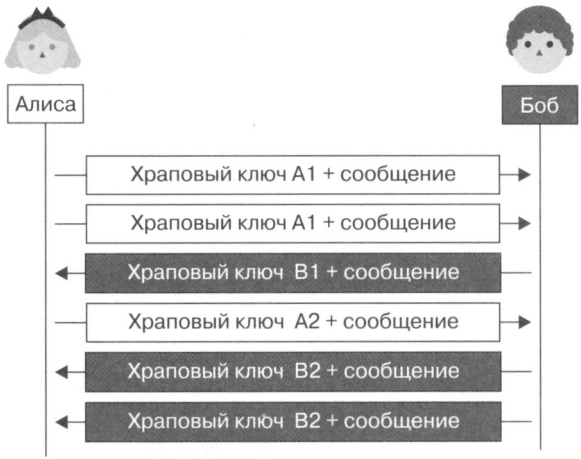


Рис. 10.16. Принцип работы храповика Диффи — Хеллмана предполагает включение открытого храпового ключа в каждое отправляемое сообщение. Этот открытый храповый ключ может совпадать с предыдущим или не совпадать в случае его обновления участником

Получив от Алисы новый храповый ключ, Боб должен выполнить обмен ключами по протоколу Диффи — Хеллмана с помощью нового храпового ключа Алисы и собственного храпового ключа. Затем результат этого обмена может применяться с симметричным храповиком для расшифровки полученных сообщений (рис. 10.17).

При получении нового храпового ключа Боб также должен сгенерировать для себя новый случайный храповый ключ, чтобы выполнить еще один обмен ключами с применением нового храпового ключа Алисы, с помощью которого он затем сможет шифровать адресованные ей сообщения (рис. 10.18).

В спецификации алгоритма двойного храповика такая последовательность обменов ключами уподобляется игре в пинг-понг.

> Поочередная замена пар храповых ключей напоминает игру в пинг-понг. Злоумышленник, скомпрометировавший ключ одной из сторон, может узнать значение текущего закрытого храпового ключа, однако этот закрытый ключ в конце концов будет заменен на нескомпрометированный. В этот момент алгоритм Диффи — Хеллмана вычислит новое неизвестное злоумышленнику выходное значение.
>
> *Спецификация алгоритма двойного храповика*

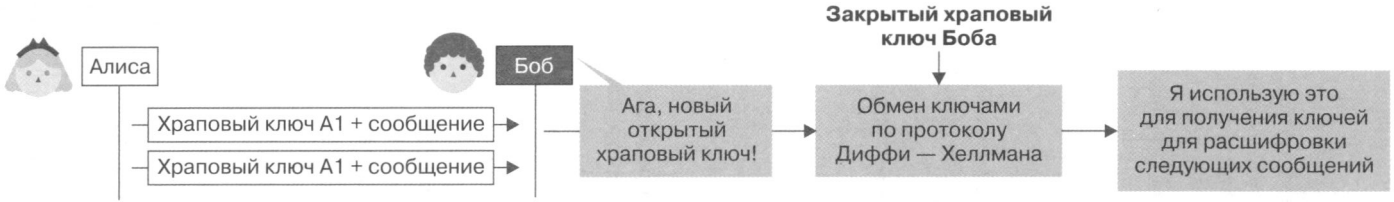


Рис. 10.17. Получив от Алисы новый открытый храповый ключ, Боб должен выполнить обмен ключами с использованием нового ключа Алисы и собственного храпового ключа, чтобы получить ключи для расшифровки последующих сообщений Алисы с помощью симметричного храповика



Рис. 10.18. Дополнение к схеме, приведенной на рис. 10.17. После получения нового храпового ключа Боб также должен сгенерировать новый храповый ключ для себя. Он будет использоваться для получения ключей шифрования и включаться в последующие адресованные Алисе сообщения до тех пор, пока он не получит от нее новый храповый ключ

Комбинация DH-храповика и симметричного храповика называется *двойным храповиком*. На рис. 10.19 я попытался представить это в виде одной диаграммы.

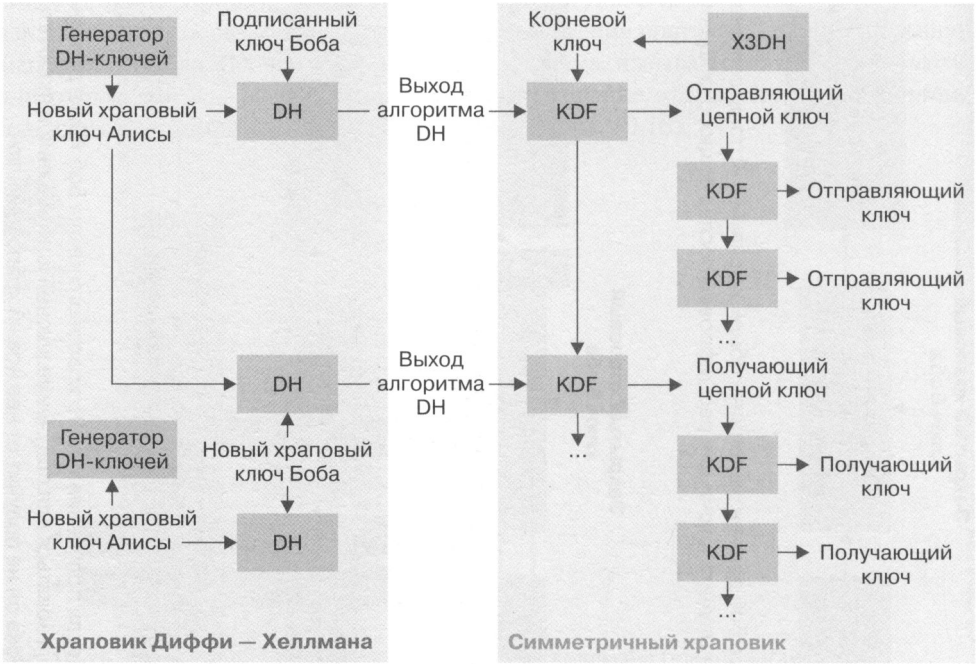


Рис. 10.19. Если смотреть с точки зрения Алисы, то двойной храповик представляет собой комбинацию храповика Диффи — Хеллмана *(слева)* и симметричного храповика *(справа)*. Это обеспечивает безопасность после компрометации, а также прямую секретность протокола, используемого после X3DH-рукопожатия. При отправке первого сообщения Алисе еще неизвестен храповый ключ Боба, поэтому вместо него она применяет подписанный им ключ

Я знаю, что эта диаграмма довольно сложна для восприятия, поэтому, если вы хотите получить более подробное описание принципа работы протокола Signal, рекомендую ознакомиться с его спецификацией по адресу https://signal.org/docs.

10.5. ТЕКУЩЕЕ СОСТОЯНИЕ ОБЛАСТИ СКВОЗНОГО ШИФРОВАНИЯ

Сегодня большая часть безопасных коммуникаций между пользователями осуществляется не через зашифрованную электронную почту, а через приложения для безопасного обмена сообщениями. Протокол Signal уверенно победил в своей категории и был внедрен во многие проприетарные приложения и такие федеративные протоколы, находящиеся в открытом доступе, как XMPP (через расширение OMEMO) и Matrix (современная альтернатива IRC). В то же время протоколы

PGP и S/MIME постепенно утрачивают популярность в связи с потерей доверия, обусловленной успешными атаками на них.

Что же делать, если вы хотите написать собственное приложение для обмена сообщениями со сквозным шифрованием? К сожалению, многое из того, что сейчас используется в данной области, представляет собой решения, разработанные для конкретного случая, так что для получения полнофункциональной и безопасной системы вам придется самостоятельно дорабатывать многие детали. Большая часть кода протокола Signal является открытой, но он недостаточно хорошо документирован и его бывает сложно применять правильно. Вам может больше повезти с такими открытыми децентрализованными решениями, как Matrix, интегрировать которые гораздо проще. Именно так поступило французское правительство.

В завершение этой главы я хотел бы рассказать о ряде открытых вопросов и проблем, требующих активного изучения. Среди прочего к ним относятся:

- групповой обмен сообщениями;
- поддержка нескольких устройств;
- предоставление лучших гарантий безопасности по сравнению с TOFU.

Начнем с первого пункта — *группового обмена сообщениями*. В разных приложениях он реализован по-разному, но, несмотря на это, по-прежнему остается предметом активного изучения. Например, приложение Signal предполагает такое использование групповых чатов, при котором серверам оказываются видны только пары общающихся друг с другом пользователей — ни больше ни меньше. Это означает, что клиентам приходится зашифровывать и отправлять сообщение группового чата каждому из его участников по отдельности. Этот подход называется *веерной рассылкой со стороны клиента* (client-side fanout) и не очень хорошо масштабируется. Кроме того, при его использовании сервер может легко вычислить участников группы, если, например, увидит, что Алиса отправляет несколько сообщений одинаковой длины Бобу и Чарльзу (рис. 10.20).

В то же время приложение WhatsApp применяет вариант протокола Signal, позволяющий серверу установить членство пользователя в групповом чате. Это дает участнику возможность отправить одно зашифрованное сообщение на сервер, который, в свою очередь, перешлет его всем членам группы. Данный подход называется *веерной рассылкой со стороны сервера* (server-side fanout).

Еще одной проблемой группового чата является *масштабирование*, то есть обеспечение безопасного обмена сообщениями в группах с большим количеством участников. Для ее решения был разработан стандарт *Messaging Layer Security* (MLS). Однако в этом направлении предстоит проделать еще много работы. Да и можно ли говорить о какой-то конфиденциальности в групповом чате, количество участников в котором превышает сотню?

ПРИМЕЧАНИЕ

В этой области все еще ведутся активные исследования, и разные подходы предполагают различные компромиссы в плане безопасности и удобства использования. Например, в 2021 году ни один протокол группового чата, судя по всему, не обеспечивал такого криптографического свойства, как *согласованность транскрипта*, гарантирующего, что все участники группового чата видят одни и те же сообщения в одном и том же порядке.

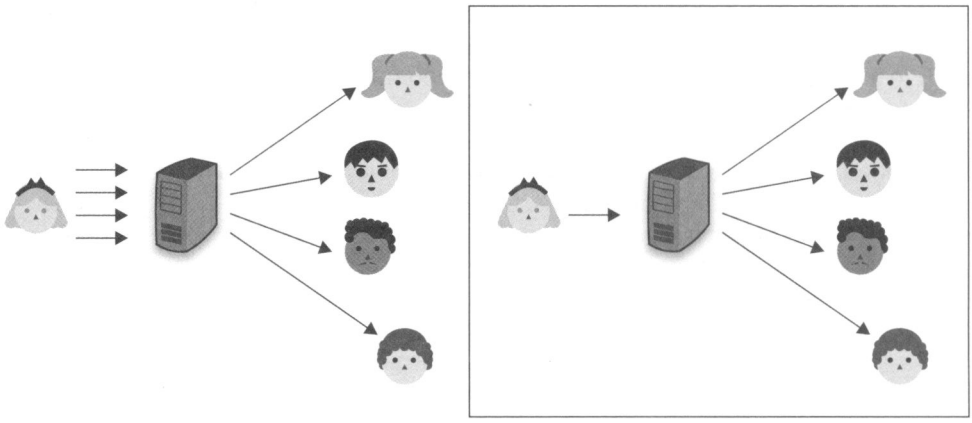


Рис. 10.20. Существует два способа сквозного шифрования групповых чатов. Веерная рассылка со стороны клиента означает, что он должен отправлять сообщения каждому из получателей по отдельности, задействуя уже существующий зашифрованный канал. Это хороший подход, если вам нужно скрыть от сервера членство пользователей в группе. Посредством веерной рассылки сервер пересылает сообщение каждому из участников группового чата. С точки зрения клиента это хороший подход для уменьшения количества отправляемых сообщений

Поддержка нескольких устройств либо вообще отсутствует, либо реализована различными способами, чаще всего предполагающими восприятие множества ваших устройств в качестве разных участников группового чата. Модель TOFU способна значительно усложнить работу с несколькими устройствами, поскольку управление разными идентификационными ключами для каждого из устройств может представлять огромную проблему. Представьте, что вам пришлось бы проверять отпечатки пальцев у каждого из своих устройств и каждого из устройств друзей. Протокол Matrix, например, позволяет пользователю подписывать собственные устройства. После проверки связанной с ними подписи другие пользователи могут доверять всем вашим устройствам как единой сущности.

Наконец, я упомянул о том, что модель TOFU тоже неидеальна, поскольку она основана на проявлении доверия к открытому ключу при первом столкновении с ним, а большинство пользователей впоследствии не проверяют отпечатки пальцев на предмет их совпадения. Можно ли с этим что-то сделать? Что, если сервер решит выдать себя за Боба только при взаимодействии с Алисой? Эту проблему призван решить протокол *Key Transparency*. Он был предложен компанией Google и напоминает протокол Certificate Transparency, о котором шла речь в главе 9. Помимо этого, проводятся исследования с помощью технологии блокчейн, о которой мы поговорим в главе 12, посвященной криптовалютам.

РЕЗЮМЕ

- Сквозное шифрование предназначено для защиты коммуникаций между реальными людьми. Протокол, использующий сквозное шифрование, более устойчив к уязвимостям, связанным с серверами-посредниками, и может существенно упростить юридические требования, предъявляемые к компаниям.
- Системам сквозного шифрования необходим способ установления доверия между пользователями. Для этого может применяться уже известный им открытый ключ или внеполосный канал, которому они доверяют.
- PGP и S/MIME — это основные протоколы, используемые сегодня для шифрования электронной почты, однако ни один из них не считается безопасным из-за применения старых криптографических алгоритмов и методов. Кроме того, они плохо интегрируются с почтовыми клиентами, что на практике означает уязвимость к различным видам атак.
  - PGP задействует модель сети доверия (WOT), в которой пользователи подписывают открытые ключи друг друга, чтобы другие пользователи могли им доверять.
  - S/MIME использует инфраструктуру открытых ключей для установления доверия между участниками. Чаще всего этот протокол применяется в компаниях и университетах.
- Альтернатива PGP — протокол saltpack, который устраняет ряд проблем и предполагает использование социальных сетей для обнаружения открытых ключей других людей.
- В электронной почте всегда будут возникать проблемы с шифрованием, поскольку данный протокол изначально этого не предполагал. В то же время современные протоколы и приложения для обмена сообщениями считаются лучшей альтернативой зашифрованным электронным письмам, поскольку они разработаны с учетом сквозного шифрования.
  - Протокол Signal используется большинством приложений для обмена сообщениями с целью сквозного шифрования коммуникаций между пользователями. К таким приложениям относятся мессенджеры Signal, WhatsApp, Facebook Messenger и Skype.
  - Другие протоколы, например Matrix, призваны стандартизировать федеративные протоколы для сквозного шифрования процесса обмена сообщениями. Федеративные протоколы — это открытые протоколы, с которыми может взаимодействовать кто угодно, в отличие от централизованных протоколов, которые применяются лишь одним приложением.

11 Аутентификация пользователей

В этой главе

- ✓ Разница между аутентификацией людей и данных.
- ✓ Аутентификация пользователей с помощью паролей и ключей.
- ✓ Управляемая пользователем аутентификация для защиты соединений между его устройствами.

Во введении к книге я свел всю криптографию к таким двум понятиям, как конфиденциальность и аутентификация. В реальных приложениях конфиденциальность обычно приносит наименьшие проблемы, а наибольшие сложности связаны с аутентификацией. Я знаю, что уже много говорил об аутентификации, однако это понятие может вызывать путаницу, поскольку в криптографии оно имеет несколько значений. По этой причине я начинаю данную главу с объяснения того, что же аутентификация представляет собой на самом деле. Как обычно бывает в криптографии, ни один протокол не универсален, поэтому далее я познакомлю вас со множеством протоколов аутентификации, используемых в реальных приложениях.

11.1. КРАТКИЙ ОБЗОР ТЕМЫ АУТЕНТИФИКАЦИИ

В предыдущих главах вы уже много раз слышали об аутентификации, так что давайте кратко пробежимся по изученному материалу. Итак, вы узнали:

- об аутентификации в таких криптографических примитивах, как коды аутентификации сообщений (см. главу 3) и аутентифицированное шифрование (см. главу 4);
- аутентификации в таких криптографических протоколах, как TLS (см. главу 9) и Signal (см. главу 10), позволяющих аутентифицировать одного или нескольких участников.

В первом случае под аутентификацией понимается установление *подлинности* (или *целостности*) сообщений, а во втором — *подтверждение своей личности при взаимодействии с кем-то другим*. Обозначение разных понятий одним и тем же словом может сбить с толку! Однако, согласно Оксфордскому словарю английского языка (http://www.oed.com/), оба варианта употребления правильные: *«Аутентификация. Процесс или действие, связанные с доказательством или демонстрацией того, что нечто является истинным, подлинным или действительным»*.

Поэтому помните о том, что в криптографии термином «аутентификация» могут обозначаться два различных понятия в зависимости от контекста.

- *Аутентификация сообщения/рабочей нагрузки*. В данном случае вы доказываете, что сообщение является подлинным и не изменялось с момента создания. (Например, аутентифицированы ли эти сообщения или кто-то мог их подделать?)
- *Аутентификация происхождения/сущности/личности*. В данном случае вы доказываете, что объект действительно является тем, за кого себя выдает. (Например, действительно ли я взаимодействую с сайтом google.com?)

Таким образом, аутентификация предполагает доказательство того, что нечто является тем, чем оно и должно быть, и этим нечто может быть человек, сообщение или что-то еще. В этой главе под аутентификацией я буду подразумевать исключительно идентификацию людей или машин. Другими словами, в ней речь пойдет об аутентификации *личности*. Кстати, вы уже не раз сталкивались с этим типом аутентификации.

- В главе 9, посвященной безопасной передаче данных, узнали о том, что машины могут выполнять аутентификацию других машин с помощью инфраструктуры открытых ключей (ИОК).
- В главе 10, посвященной сквозному шифрованию, узнали о том, как люди могут аутентифицировать личности друг друга с помощью подхода TOFU (доверие при первом использовании с последующей проверкой) или сети доверия (WOT).

В этой главе вы познакомитесь с двумя другими случаями (рис. 11.1):

- с *аутентификацией пользователей*, или тем, как машины устанавливают личности людей;
- *управляемой пользователем аутентификацией*, или тем, как люди могут помочь машинам аутентифицировать друг друга.

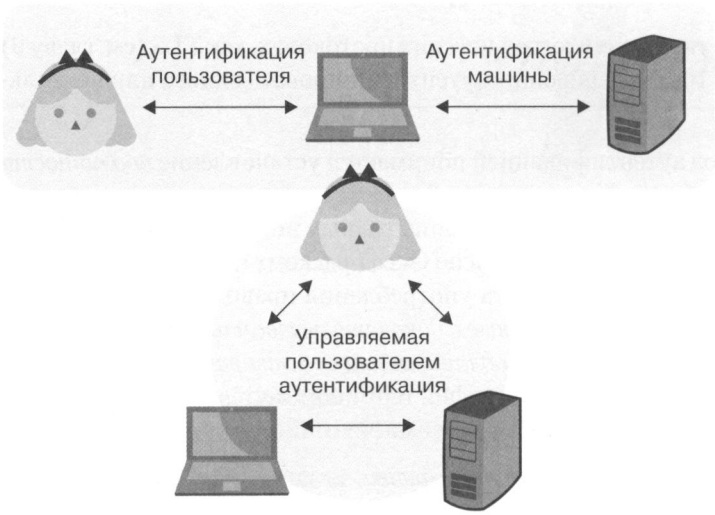


Рис. 11.1. В книге речь идет о трех типах сценариев аутентификации. При аутентификации пользователя некое устройство устанавливает личность человека. При аутентификации машины одна машина аутентифицирует другую. Управляемая пользователем аутентификация предполагает участие человека в процессе аутентификации одной машины другой машиной

Еще одним аспектом аутентификации является понятие личности. В самом деле, как можно определить кого-то вроде Алисы в криптографическом протоколе? Как машина может проверить вашу или мою личность? К сожалению (или к счастью), между плотью и битами существует непреодолимый разрыв. Для установления связи между реальным и цифровым мирами мы всегда предполагаем, что Алиса является единственным человеком, обладающим некоторыми секретными данными, знание которых она должна продемонстрировать, чтобы подтвердить свою личность. Для этого она может, например, ввести свой пароль или подписать случайный вызов, используя закрытый ключ, связанный с ее открытым ключом.

На этом вступление можно завершить. Если что-то в этом разделе показалось вам непонятным, описанные далее примеры все прояснят. Для начала рассмотрим многочисленные способы, с помощью которых машины аутентифицируют нас, людей!

11.2. АУТЕНТИФИКАЦИЯ ПОЛЬЗОВАТЕЛЕЙ, ИЛИ ПОПЫТКА ИЗБАВИТЬСЯ ОТ ПАРОЛЕЙ

Первая часть этой главы посвящена тому, как машины аутентифицируют людей, то есть *аутентификации пользователей*. Существует множество способов сделать это, но ни один из них не универсален. Однако в большинстве сценариев аутентификации пользователей мы исходим из следующего:

- сервер уже прошел аутентификацию;
- пользователь установил с ним защищенное соединение.

Например, можем представить, что сервер аутентифицирован с помощью веб-инфраструктуры открытых ключей (веб-ИОК), а соединение защищено с помощью протокола TLS (и то и другое рассматривается в главе 9). В некотором смысле большая часть этого раздела посвящена преобразованию соединения с односторонней аутентификацией в соединение с взаимной аутентификацией (рис. 11.2).



Рис. 11.2. Аутентификация пользователя обычно происходит по уже защищенному каналу, но аутентифицирован при этом только сервер. Типичным примером является случай, когда вы просматриваете веб-страницы с помощью протокола HTTPS и заходите на сайт с помощью своих учетных данных

Должен вас предупредить, что аутентификация пользователей — это огромная страна нарушенных обещаний. Вы наверняка много раз применяли пароли для аутентификации на различных веб-ресурсах, и ваш опыт, скорее всего, выглядит примерно так.

1. Вы регистрируетесь на сайте, указывая имя пользователя и пароль.
2. Заходите на сайт с помощью своих новых учетных данных.
3. Меняете пароль после восстановления учетной записи или по требованию сайта.
4. Если вам не повезло, ваш пароль (или его хеш) становится известен третьим лицам в результате серии утечек баз данных.

Знакомо?

ПРИМЕЧАНИЕ

В этой главе мы не будем касаться темы восстановления паролей/учетных записей, поскольку она имеет мало общего с криптографией. Просто знайте, что их восстановление часто связано со способом регистрации. Например, если вы зарегистрировались с помощью сотрудников ИТ-отдела своей компании, то вам, скорее всего, придется обратиться к ним в случае потери пароля, и они могут оказаться самым слабым звеном в системе защиты, если вы не будете проявлять осторожность. В самом деле, если я могу восстановить доступ к вашей учетной записи, позвонив по номеру и назвав дату вашего рождения, то вам не поможет даже самый надежный криптографический алгоритм.

Наивный способ реализации описанного процесса аутентификации пользователя предполагает сохранение его пароля при регистрации и запрос при каждом входе в систему. Как говорилось в главе 3, после успешной аутентификации пользователь обычно получает файл cookie, который может отправлять при каждом последующем запросе вместо имени пользователя и пароля. Но подождите, если сервер хранит ваш пароль в открытом виде, то при любом взломе его баз данных злоумышленники смогут узнать его, а затем применять для входа на любые сайты, при регистрации на которых вы указали тот же пароль.

Более предпочтительный способ хранения паролей — использование алгоритма *хеширования паролей* наподобие стандартизированного Argon2, о котором говорилось в главе 2. Это позволяет предотвратить утечку вашего пароля в случае взлома базы данных, правда, злоупотребляющий гостеприимством хакер все равно сможет видеть пароль при каждом вашем входе в систему. Тем не менее многие сайты и компании по-прежнему хранят пароли в открытом виде.

УПРАЖНЕНИЕ

Иногда приложения пытаются решить проблему, связанную с тем, что сервер узнает пароль пользователя при регистрации, позволяя клиенту хешировать пароль перед отправкой на сервер. Можете ли вы определить, действительно ли это работает?

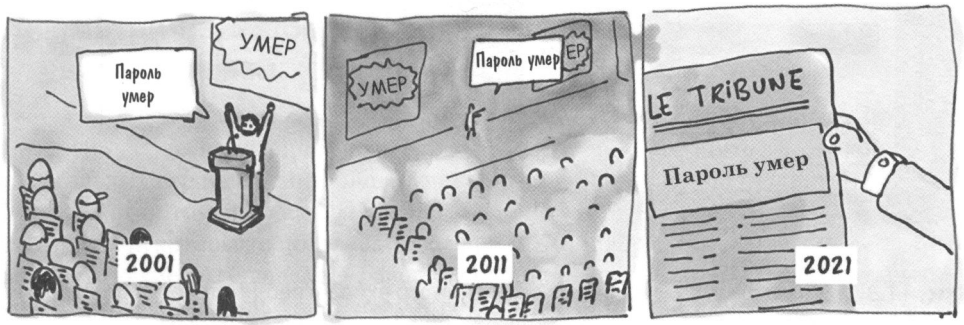
Ситуация усугубляется еще и тем, что людям удобнее иметь дело с небольшими и легко запоминающимися паролями. И они по возможности стремятся использовать один и тот же пароль повсюду.

> В 81 % всех случаев взлома используются украденные или слабые пароли.
>
> *Отчет компании Verizon об утечках данных (2017)*

Проблема слабых паролей и их многократного применения привела к появлению множества смехотворных и раздражающих паттернов дизайна, принуждающих пользователей относиться к паролям более серьезно. Например, некоторые сайты требуют использовать в паролях специальные символы, менять пароль каждые шесть месяцев и т. д. Кроме того, многие протоколы пытаются «исправить» пароли

или вовсе избавиться от них. Каждый год эксперты по безопасности заявляют о смерти концепции пароля. Тем не менее он по-прежнему остается самым распространенным механизмом аутентификации пользователей.

Как вы понимаете, пароли, скорее всего, не исчезнут в ближайшем будущем. Однако существует множество протоколов, которые улучшают или заменяют пароли. Давайте рассмотрим их.



11.2.1. Один пароль для всего. Технология единого входа и менеджеры паролей

Итак, многократное использование паролей — это плохая идея, но что же нам делать? Наивные пользователи могут придумать разные пароли для разных сайтов, но у такого подхода есть две проблемы:

- чрезмерная сложность создания большого количества разных паролей;
- лишняя умственная нагрузка, связанная с запоминанием нескольких паролей.

Для решения этих проблем широко используются два решения.

- *Технология единого входа* (single sign-on, SSO). Суть данной идеи заключается в том, что пользователь может подключиться к множеству различных сервисов, подтвердив факт наличия у него учетной записи какого-то одного сервиса. Таким образом, пользователю необходимо запомнить только тот пароль, который связан с этим конкретным сервисом, для подключения ко всем остальным. Примером может стать кнопка Continue with Facebook (Продолжить с Facebook), показанная на рис. 11.3.
- *Менеджеры паролей*. Технология SSO удобна, если все используемые вами сервисы ее поддерживают, однако этот подход явно не работает в масштабе всей Всемирной паутины. В этом случае более предпочтительный подход — совершенствование клиентов, а не попытка решить проблему на стороне сервера. В современных браузерах есть встроенные менеджеры паролей, которые могут предлагать сложные комбинации символов при регистрации на новых сайтах и хранить все ваши пароли. При этом вам необходимо запомнить лишь один мастер-пароль.

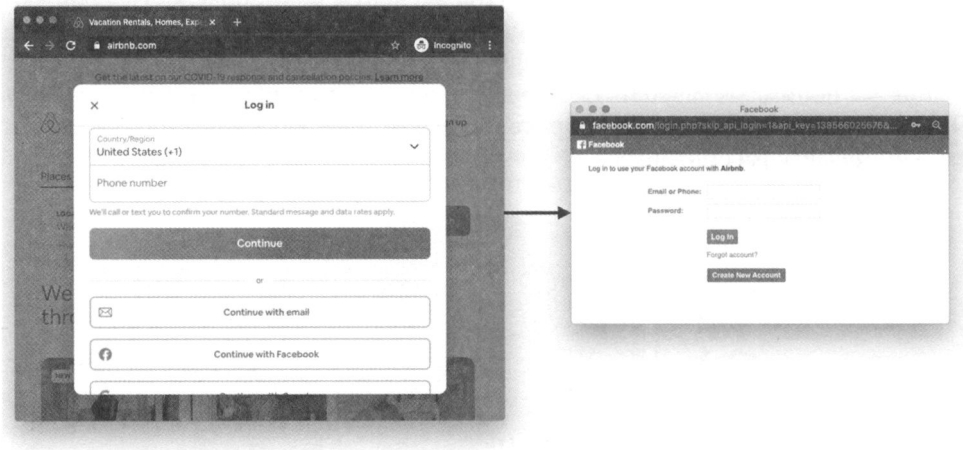


Рис. 11.3. Пример применения технологии единого входа в веб-контексте. При наличии учетной записи Facebook или Google пользователь может подключаться к другим сервисам (в данном случае — Airbnb), не придумывая новый пароль

Концепция SSO уже некоторое время применяется в корпоративном мире, но обычным пользователям она стала доступна относительно недавно. Сегодня основными протоколами, реализующими технологию единого входа, являются:

- *SAML* (Security Assertion Markup Language 2.0 — язык разметки декларации безопасности) — протокол, использующий кодировку XML (Extensible Markup Language — расширяемый язык разметки);
- *OIDC* (OpenID Connect) — расширение протокола авторизации OAuth 2.0 (RFC 6749), применяющее кодировку JSON (JavaScript Object Notation — нотация объектов JavaScript).

Протокол SAML по-прежнему широко используется в основном в корпоративной среде, но считается устаревшим. В то же время OIDC можно встретить во всевозможных веб- и мобильных приложениях. Скорее всего, вы его уже применяете!

Протоколы аутентификации часто считаются труднореализуемыми. Например, протокол OAuth2, на который опирается OIDC, печально известен тем, что предоставляет возможности для злоупотребления. А вот протокол OIDC имеет хорошую спецификацию (https://openid.net). Старайтесь следовать стандартам и применять лучшие практики — это может избавить вас от многих проблем.

ПРИМЕЧАНИЕ

Вот еще один пример того, что может произойти с весьма крупной компанией, решившей *не* следовать этому совету. В мае 2020 года в системе авторизации через Apple (sign-in with Apple), использующей стандарт OIDC, была выявлена уязвимость, позволяющая любому человеку получить действительный ID-токен для любой учетной записи Apple, просто запросив его у серверов компании.

Технология SSO очень удобна для пользователей, поскольку сокращает количество паролей, с которыми им приходится иметь дело, но не устраняет пароли полностью. Пользователю по-прежнему приходится задействовать их для подключения к OIDC-провайдерам. Далее мы поговорим о способах сокрытия паролей с помощью криптографии.

11.2.2. Не хотите видеть их пароли? Используйте асимметричный обмен ключами с парольной аутентификацией

В предыдущем разделе мы говорили о решениях, которые упрощают процесс управления идентификацией, позволяя пользователям проходить аутентификацию в нескольких сервисах с помощью учетной записи, связанной с одним сервисом. Хотя такие протоколы, как OIDC, весьма полезны, поскольку уменьшают количество паролей, которыми пользователям приходится управлять, они не меняют того факта, что какие-то сервисы все равно могут видеть пароль пользователя в открытом виде. Даже если пароль хранится у них как результат хеширования, он все равно отправляется в открытом виде каждый раз, когда пользователь регистрируется, меняет пароль или входит в систему.

Семейство криптографических протоколов под названием *«асимметричный (или дополненный) обмен ключами с парольной аутентификацией»* (password-authenticated key exchange, PAKE) призвано обеспечить аутентификацию пользователей, при которой им не приходится сообщать пароли напрямую серверу. Это отличается от *симметричных или сбалансированных протоколов PAKE*, при использовании которых пароль известен обеим сторонам.

Самым популярным асимметричным PAKE-протоколом на данный момент является *Secure Remote Password* (SRP), который был стандартизирован в 2000 году (RFC 2944, Telnet Authentication: SRP) и позднее интегрирован в TLS (RFC 5054, Using the Secure Remote Password (SRP) Protocol for TLS Authentication). Это довольно старый протокол, который имеет ряд недостатков. Например, если злоумышленник, реализующий атаку посредника, перехватит поток регистрации, то он сможет выдать себя за жертву и войти в систему от ее имени. Также он плохо сочетается с современными протоколами, поскольку не может быть реализован на основе эллиптических кривых и, что еще хуже, несовместим с TLS 1.3.

С момента изобретения протокола SRP было предложено и стандартизировано множество асимметричных PAKE-протоколов. Летом 2019 года исследовательская группа Crypto Forum Research Group (CFRG) Инженерного совета Интернета приступила к процессу отбора PAKE-протоколов, задавшись целью стандартизировать один алгоритм для каждой из двух категорий таких протоколов: симметричных/сбалансированных и асимметричных/дополненных. В марте 2020 года группа CFRG объявила о том, что ее выбор пал на следующие протоколы:

- *CPace* — рекомендуемый симметричный/сбалансированный PAKE-протокол, изобретенный Бьорном Хаазе и Бенуа Лабриком;
- *OPAQUE* — рекомендуемый асимметричный/дополненный PAKE-протокол, разработанный Станиславом Ярецки, Хьюго Кравчиком и Цзяюй Сюем.

В этом разделе я расскажу о протоколе OPAQUE, который в начале 2021 года все еще находился в процессе стандартизации. Позже вы узнаете больше и о симметричных PAKE-протоколах и CPace.

Название OPAQUE происходит от омонима *O-PAKE*, где буква *O* означает *oblivious* (забывчивый). Это связано с тем, что протокол OPAQUE опирается на такой криптографический примитив, как *забывчивая псевдослучайная функция* (oblivious pseudorandom function, OPRF), о которой я еще не упоминал.

Забывчивые псевдослучайные функции. OPRF — это протокол с двумя участниками, который имитирует работу псевдослучайных функций (PRF), речь о которых шла в главе 3. Напомню, что PRF в некоторой степени эквивалентна имитовставке (MAC): она принимает ключ и входные данные и выдает совершенно случайное значение фиксированной длины.

ПРИМЕЧАНИЕ

В криптографии термин oblivious («забывчивый») обычно применяется к протоколам, в которых одна сторона вычисляет результат криптографической операции, не зная входных данных, предоставленных другой стороной.

Вот как работает функция OPRF в общих чертах.

1. Алиса хочет вычислить значение PRF на основе входных данных, сохранив их в секрете. Для этого она «ослепляет» свои входные данные с помощью случайного значения, называемого *фактором ослепления*, и отправляет полученный результат Бобу.
2. Боб применяет OPRF к этому ослепленному значению с помощью своего секретного ключа, но результат все равно остается ослепленным, а значит, бесполезным для Боба. Он отправляет его обратно Алисе.
3. Алиса применяет к ослепленному результату тот же фактор ослепления, которым пользовалась ранее, чтобы получить реальный результат.

Важно отметить, что при каждом применении этого протокола Алисе приходится создавать новый фактор ослепления. Однако вне зависимости от этого фактора при одних и тех же входных данных она всегда будет получать один и тот же результат (рис. 11.4).

Вот пример протокола OPRF, реализованного в группе, где задача дискретного логарифмирования оказывается трудной.

1. Алиса преобразует свои входные данные в элемент группы x.
2. Алиса генерирует случайный фактор ослепления r.
3. Алиса ослепляет свои входные данные, вычисляя значение $ослепленный_вход = x^r$.
4. Алиса посылает *ослепленный_вход* Бобу.
5. Боб вычисляет значение $ослепленный_выход = ослепленный_вход^k$, где k — секретный ключ.
6. Боб отправляет результат обратно Алисе.
7. Алиса отменяет ослепление полученного результата, вычисляя значение $выход = ослепленный_выход^{1/r} = x^k$, где $1/r$ — величина, обратная r.

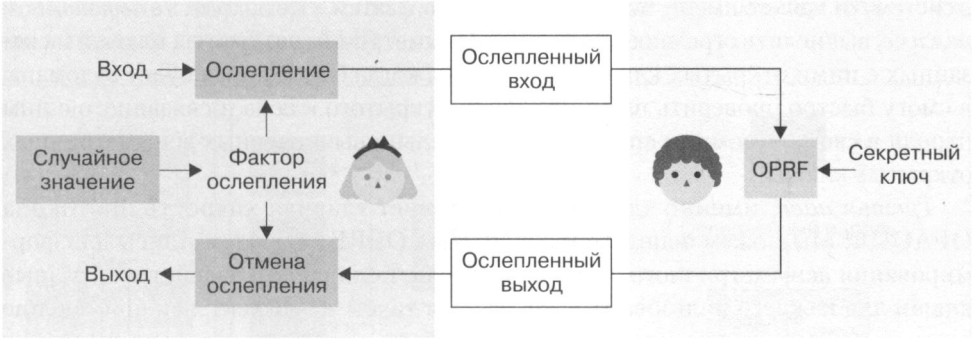


Рис. 11.4. Забывчивая PRF — это конструкция, которая позволяет одной стороне вычислить значение PRF на основе входных данных другой стороны, не зная их. Для этого Алиса сначала генерирует случайный фактор ослепления, применяет его к своим входным данным и отправляет результат Бобу. Используя свой секретный ключ, Боб применяет PRF к полученному ослепленному значению и отправляет ослепленные выходные данные Алисе, которая может отменить ослепление. При этом результат не зависит от значения фактора ослепления

В том, как OPAQUE использует эту интересную конструкцию, и заключается весь фокус асимметричного PAKE-протокола.

Принцип работы асимметричного PAKE-протокола OPAQUE. Суть заключается в том, чтобы обеспечить клиенту, например Алисе, возможность выполнить аутентифицированный обмен ключами с неким сервером. Мы предполагаем, что Алиса уже знает открытый ключ этого сервера или может его аутентифицировать (если в качестве сервера выступает HTTPS-сайт, она может использовать для этого веб-ИОК). Далее последовательно рассмотрим все аспекты работы протокола OPAQUE.

Первая идея заключается в применении криптографии с открытым ключом для аутентификации соединения на стороне Алисы. Если Алиса владеет парой долгосрочных ключей, а сервер знает открытый ключ, то она может просто использовать свой закрытый ключ для обмена ключами с сервером с взаимной аутентификацией или подписать отправленный сервером вызов. К сожалению, асимметричный закрытый ключ слишком длинный, и Алиса помнит только свой пароль. Она могла бы хранить пару ключей на своем рабочем устройстве, но хочет иметь возможность позднее войти в систему с другого устройства.

Вторая идея: Алиса может получить асимметричный закрытый ключ из своего пароля, используя функцию формирования ключа (KDF) на основе пароля наподобие Argon2, о которой мы говорили в главах 2 и 8. После этого открытый ключ Алисы может храниться на сервере. Если мы хотим избежать проверки пароля по всей базе данных в случае ее взлома, то можем попросить сервер предоставить каждому пользователю уникальное значение соли, которое он сможет применять вместе с функцией KDF на основе пароля.

Это неплохо, но протокол OPAQUE направлен также на предотвращение так называемой *атаки предварительным вычислением*. Я могу попытаться войти

в систему от вашего имени, получить вашу соль, а затем, уже находясь *в автономном режиме*, вычислить огромное количество асимметричных закрытых ключей и связанных с ними открытых ключей. В тот день, когда база данных будет взломана, я смогу быстро проверить наличие вашего открытого ключа и связанного с ним пароля в своем огромном списке предварительно вычисленных асимметричных открытых ключей.

Третья идея: именно здесь в дело вступает главная хитрость протокола OPAQUE! Мы можем использовать протокол OPRF с паролем Алисы для формирования асимметричного закрытого ключа. Если сервер задействует разные ключи для каждого пользователя, то это дает такой же эффект, как применение соли, поскольку в этом случае атаки могут быть направлены только на одного пользователя единовременно. Таким образом, злоумышленнику, который хочет предварительно вычислить асимметричные закрытые ключи, делая попытки угадать наш пароль, придется выполнять *онлайн*-запросы, что не позволит ему реализовать офлайновые атаки методом перебора. Онлайн-запросы хороши тем, что их частоту можно ограничить (например, не допускать более десяти попыток входа в систему за один час), чтобы предотвратить подобные онлайн-атаки методом грубой силы.

Учтите, что на самом деле протокол OPAQUE работает несколько иначе: вместо формирования асимметричного закрытого ключа пользователь формирует симметричный ключ, который затем применяется для шифрования резервной копии вашей пары асимметричных ключей и некоторых дополнительных данных, которые могут включать, например, открытый ключ сервера (рис. 11.5).

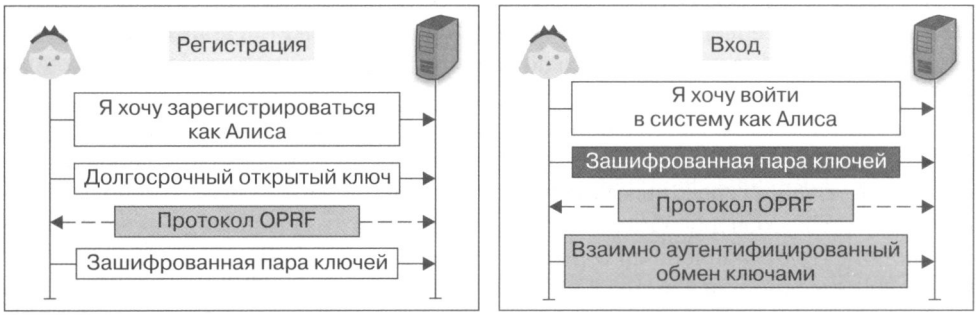


Рис. 11.5. Чтобы зарегистрироваться на сервере с помощью протокола OPAQUE, Алиса генерирует пару долгосрочных ключей и отправляет свой открытый ключ на сервер, который хранит его и связывает с личностью Алисы. Затем она задействует протокол OPRF для получения надежного симметричного ключа из своего пароля и отправляет зашифрованную резервную копию своей пары ключей на сервер. Чтобы войти в систему, Алиса получает от сервера свою зашифрованную пару ключей, затем использует протокол OPRF со своим паролем, чтобы получить симметричный ключ, способный расшифровать ее пару ключей. После этого ей остается лишь выполнить взаимно аутентифицированный обмен ключами с сервером (или подписать его вызов) с помощью этого ключа

Прежде чем переходить к следующему разделу, давайте подведем итоги того, что вы узнали в этом (рис. 11.6).

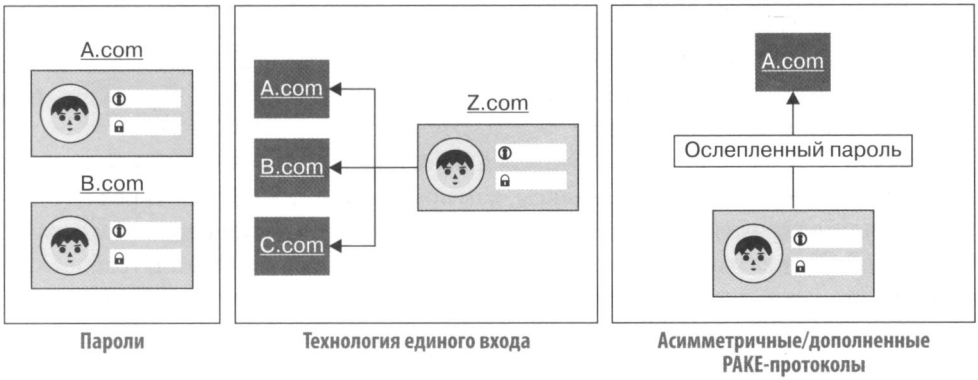


Рис. 11.6. Пароли — удобное средство аутентификации пользователей, поскольку они хранятся в памяти людей и могут использоваться на любом устройстве. В то же время людям сложно создавать надежные пароли, а поскольку они склонны применять одни и те же пароли на разных сайтах, утечка паролей может нанести им существенный вред. Технология единого входа позволяет подключаться ко многим сервисам с помощью одного или нескольких сервисов, а асимметричный (или дополненный) обмен ключами с парольной аутентификацией дает возможность пользователям аутентифицироваться так, чтобы сервер никогда не узнал их настоящий пароль

11.2.3. Одноразовые пароли — это не совсем пароли: беспарольная аутентификация с симметричными ключами

Итак, пока все хорошо. Вы узнали о различных протоколах, которые могут применяться в приложениях для аутентификации пользователей с помощью паролей. Но, как вы уже знаете, у паролей есть недостатки. Они уязвимы для атак методом грубой силы, часто используются повторно, попадают в руки третьих лиц и т. д. Что же может стать альтернативой паролям?

Ответом являются ключи! Как вы знаете, в криптографии существует два типа ключей, и они оба могут нам пригодиться:

- симметричные ключи;
- асимметричные ключи.

Решения, основанные на симметричных ключах, мы рассмотрим в этом разделе, а на асимметричных — в следующем. Давайте представим, что Алиса регистрируется на сервисе с использованием симметричного ключа, часто генерируемого сервером и передаваемого вам с помощью QR-кода. Наивный способ последующей аутентификации Алисы заключается в том, чтобы просто попросить ее прислать

свой симметричный ключ. Этот подход не очень хорош, поскольку компрометация ее секрета может дать злоумышленнику неограниченный доступ к ее аккаунту. Вместо этого Алиса может получить так называемые *одноразовые пароли* (one-time password, OTP) из симметричного ключа и отправить их вместо долгосрочного симметричного ключа. Хотя OTP — это не пароль, его название указывает на то, что его можно применять вместо пароля, предупреждая при этом, что его никогда не следует применять повторно.

Суть аутентификации пользователей на основе OTP довольно проста: ваша безопасность обеспечивается знанием, как правило, 16- или 32-байтного случайного равномерно распределенного симметричного ключа, а не паролем с низкой энтропией. Этот симметричный ключ позволяет генерировать OTP по требованию (рис. 11.7).

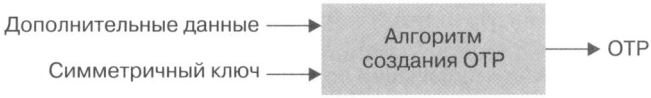


Рис. 11.7. Алгоритм создания одноразовых паролей позволяет генерировать произвольное их количество на основе симметричного ключа и некоторых дополнительных данных. Эти данные могут быть разными в зависимости от используемого алгоритма создания OTP

Аутентификация на основе OTP чаще всего реализуется в мобильных приложениях (популярный пример приведен на рис. 11.8) или ключах безопасности — небольших устройствах, которые можно подключить к USB-порту компьютера. Для создания OTP можно использовать две основные схемы.

- Алгоритм создания одноразовых паролей на основе HMAC (HOTP), стандартизированный в документе RFC 4226, представляет собой OTP-алгоритм, в котором в качестве дополнительных данных выступает счетчик.
- Алгоритм создания одноразовых паролей на основе времени (TOTP), стандартизированный в документе RFC 6238, представляет собой OTP-алгоритм, в котором в качестве дополнительных данных выступает время.

Большинство современных приложений используют TOTP, поскольку HOTP требует того, чтобы клиент и сервер сохраняли состояние (счетчик). Хранение состояния может привести к проблемам, если одна из сторон рассинхронизируется с другой и не сможет больше создавать или проверять легитимные OTP.

В большинстве случаев алгоритм TOTP работает следующим образом.

- *При регистрации сервис передает пользователю симметричный ключ (возможно, с помощью QR-кода).* Затем пользователь добавляет этот ключ в приложение TOTP.
- *При входе в систему пользователь может задействовать приложение TOTP для вычисления одноразового пароля.* Для этого вычисляется значение функции

`HMAC(symmetric_key, time)`, где `time` — текущее время, округленное до минуты для того, чтобы одноразовый пароль оставался действительным на протяжении 60 с. Затем:

- приложение TOTP отображает пользователю сформированный одноразовый пароль в усеченном и понятном человеку виде, например сокращенный до шести цифр десятичной системы счисления;
- пользователь копирует или вводит одноразовый пароль в соответствующее поле;
- приложение извлекает связанный с пользователем симметричный ключ и вычисляет одноразовый пароль так же, как это сделал пользователь. Если результат совпадает с полученным одноразовым паролем, пользователь успешно проходит аутентификацию.

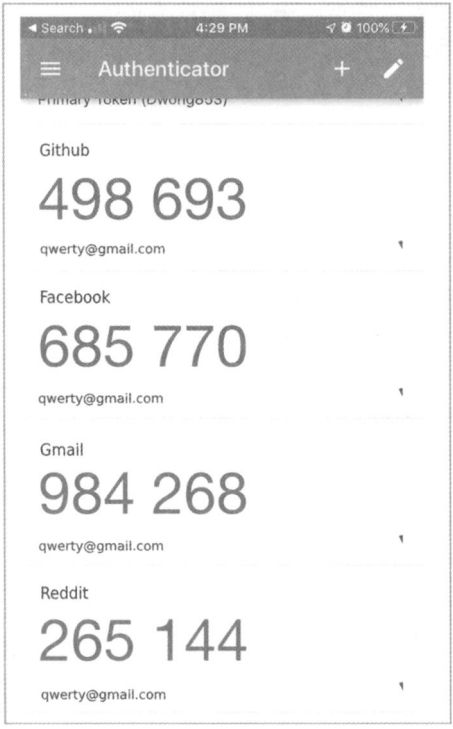


Рис. 11.8. Снимок экрана мобильного приложения Google Authenticator. Эта программа позволяет хранить уникальные для каждого приложения симметричные ключи, которые могут использоваться в рамках схемы TOTP для генерации шестизначных одноразовых паролей, остающихся действительными в течение 30 с

Разумеется, сравнение OTP пользователя и OTP, вычисленного на сервере, должно выполняться за постоянное время аналогично проверке аутентификационного тега с помощью MAC. Этот процесс продемонстрирован на рис. 11.9.

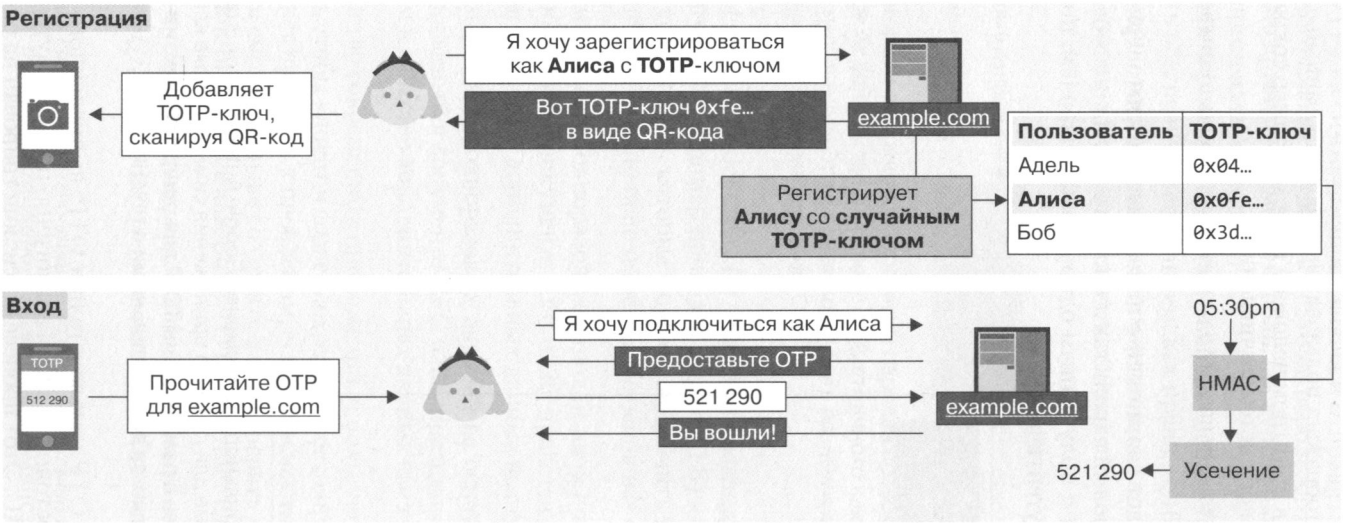


Рис. 11.9. Алиса регистрируется на сайте example.com, задействуя алгоритм TOTP для аутентификации. Она импортирует симметричный ключ с веб-сайта в свое приложение TOTP. Позднее она может попросить это приложение вычислить одноразовый пароль для example.com и использовать его для аутентификации на этом сайте. Сайт example.com извлекает симметричный ключ, связанный с Алисой, и вычисляет одноразовый пароль с помощью механизма HMAC и значения текущего времени. Затем веб-сайт сравнивает одноразовый пароль с отправленным Алисой значением за постоянное время

Однако этот способ аутентификации на основе TOTP неидеален. Некоторые из его аспектов требуют улучшения, так как:

- аутентификация может быть подделана сервером, поскольку он также владеет симметричным ключом;
- вас могут заставить выдать свой одноразовый пароль с помощью методов социальной инженерии.

По этой причине симметричные ключи являются *еще одной неидеальной заменой* паролям. Далее мы поговорим о том, как использование асимметричных ключей может устранить эти недостатки.

ФИШИНГ

Фишинг (метод *социальной инженерии*) — это атака, эксплуатирующая не уязвимости программного обеспечения, а человеческие слабости. Представьте, что приложение требует от вас ввести одноразовый пароль для прохождения аутентификации. В данном случае злоумышленник может попытаться войти в приложение под вашим именем, а когда ему будет предложено ввести одноразовый пароль, позвонить вам и, выдав себя за сотрудника компании — разработчика данного приложения, попросить назвать действительный пароль.

Вы думаете, что не попались бы на эту удочку? Хорошие социальные инженеры мастерски придумывают правдоподобные истории и создают ощущение срочности, способное заставить проболтаться даже самых стойких из нас. Если подумать, то все протоколы, о которых мы говорили ранее, уязвимы для такого рода атак.

11.2.4. Замена паролей асимметричными ключами

Когда речь идет о криптографии с открытым ключом, существует несколько способов применения асимметричных ключей для аутентификации на сервере. Мы можем:

- использовать свой асимметричный ключ в ходе обмена ключами для аутентификации нашей стороны соединения;
- использовать свой асимметричный ключ в рамках уже защищенного соединения с аутентифицированным сервером.

Рассмотрим эти способы подробнее.

Взаимная аутентификация при обмене ключами. О возможности использования асимметричного ключа в процессе обмена ключами вы уже слышали. В главе 9 я упоминал о том, что TLS-сервер в ходе выполнения рукопожатия может запросить у клиента *сертификат*. Часто компании снабжают устройства своих сотрудников

уникальным сертификатом, который позволяет им проходить аутентификацию во внутренних сервисах. На рис. 11.10 показано, как это выглядит с точки зрения пользователя.

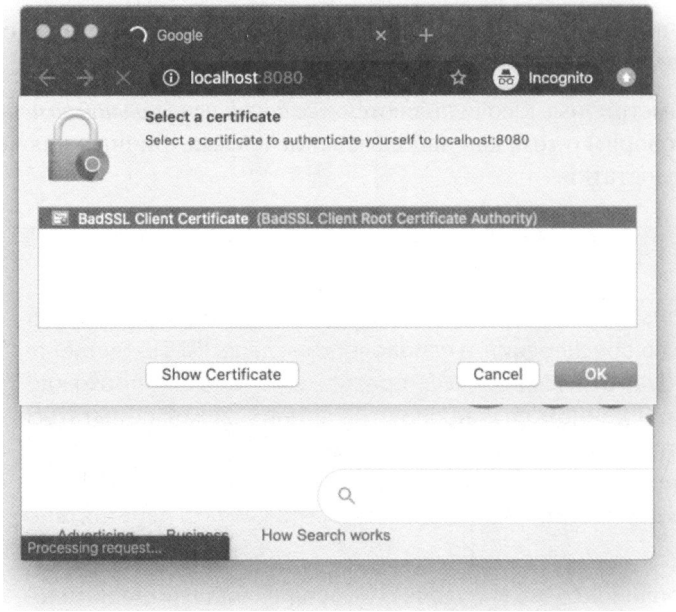


Рис. 11.10. Страница, запрашивающая у браузера пользователя сертификат клиента. Пользователь может выбрать нужный вариант из списка локально установленных сертификатов. В процессе TLS-рукопожатия ключ сертификата клиента участвует в подписании транскрипта рукопожатия, в том числе того эфемерного открытого ключа клиента, который задействуется в ходе рукопожатия

Сертификаты, применяемые на стороне клиента, довольно просты. Например, в TLS 1.3 сервер может предложить клиенту пройти аутентификацию во время рукопожатия, отправив запрос `Certificate-Request`. В ответ клиент отправляет свой сертификат в сообщении `Certificate`, за которым следует подпись всех отправленных и полученных сообщений, содержащаяся в сообщении `Certificate-Verify`, которое включает эфемерный открытый ключ, использованный при обмене ключами.

Клиент проходит аутентификацию, если серверу удается распознать сертификат и успешно проверить подпись клиента. Другим примером является протокол Secure Shell (SSH), в рамках которого клиент тоже подписывает части рукопожатия открытым ключом, известным серверу.

Обратите внимание на то, что подписание — это не единственный способ аутентификации при использовании криптографии с открытым ключом на этапе рукопожатия. Протокольный фреймворк Noise (о нем мы говорили в главе 9) предусматривает несколько схем рукопожатия, которые позволяют выполнять аутентификацию на стороне клиента посредством только обмена DH-ключами.

Аутентификация пользователей после рукопожатия с помощью FIDO2. Второй способ аутентификации с участием асимметричных ключей предполагает применение уже *защищенного* соединения, в рамках которого аутентифицируется только сервер. Для этого сервер может просто попросить клиента подписать *случайный* вызов. Это позволяет предотвратить атаки повторного воспроизведения.

Один из интересных стандартов, используемых в данной области, — *Fast IDentity Online 2* (FIDO2). Это открытый стандарт, который определяет способ применения асимметричных ключей для аутентификации пользователей. Он предназначен для защиты от фишинговых атак, поэтому работает только с *аппаратными аутентификаторами*. Так называются физические компоненты, способные генерировать и хранить ключи подписи и подписывать произвольные вызовы. Стандарт FIDO2 разбит на две спецификации (рис. 11.11).

- *CTAP* (Client to Authenticator Protocol — протокол «клиент — аутентификатор») — определяет протокол, который могут использовать *внешние* по отношению к вашему основному устройству *аутентификаторы* и *клиенты* для связи друг с другом. Клиент в спецификации CTAP определяется как программное обеспечение, которое посылает этим аутентификаторам запросы в процессе аутентификации. Таким образом, в качестве клиента может выступать операционная система, нативное приложение, например браузер, и т. д.
- *WebAuthn* (Web Authentication — веб-аутентификация) — это протокол, который могут применять веб-браузеры и веб-приложения для аутентификации пользователей с помощью аппаратных аутентификаторов. Таким образом, для обеспечения поддержки аутентификаторов данный протокол должен реализовываться браузерами. Если вы создаете веб-приложение и хотите обеспечить аутентификацию пользователей с помощью аппаратных аутентификаторов, вам следует задействовать именно протокол WebAuthn.

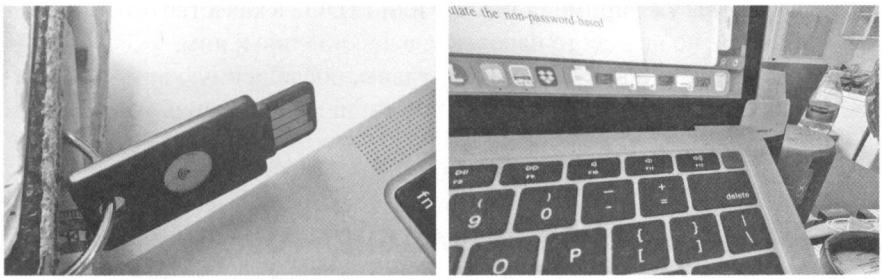


Рис. 11.11. Два типа аппаратных аутентификаторов, которые можно использовать с FIDO2: внешний аутентификатор Yubikey *(слева)* и встроенный аутентификатор TouchID *(справа)*

Протокол WebAuthn позволяет веб-сайтам использовать не только внешние, но и встроенные в ваше устройство аутентификаторы. На разных платформах они реализованы по-разному и часто защищены биометрическим устройством, например считывателем отпечатков пальцев, системой распознавания лиц и т. д.

На этом мы заканчиваем первую часть главы. Но перед тем, как переходить к следующей, резюмируем все, что узнали о беспарольных протоколах аутентификации (рис. 11.12).



Рис. 11.12. При беспарольной аутентификации приложения могут позволить пользователям применять либо симметричные ключи в рамках протоколов на основе OTP, либо асимметричные ключи при использовании стандарта FIDO2. Этот стандарт поддерживает различные типы аутентификаторов: внешние (через протокол CTAP) и встроенные

Теперь, когда вы познакомились со множеством различных методов и протоколов, позволяющих усовершенствовать пароли или заменить их более надежными криптографическими решениями, вы, возможно, задаетесь вопросом: какие из них следует использовать? У каждого из этих решений есть свои недостатки, и ни одно из них не является всеобъемлющим. Однако вы можете комбинировать их! Эта идея называется *многофакторной аутентификацией* (MFA). На самом деле вполне вероятно, что вы уже применяли OTP или FIDO2 в качестве второго фактора аутентификации, но не вместо паролей, а в дополнение к ним.

На этом мы завершаем первую половину главы, посвященную аутентификации пользователей. Далее поговорим о том, как люди могут помочь устройствам аутентифицировать друг друга.

11.3. АУТЕНТИФИКАЦИЯ, УПРАВЛЯЕМАЯ ПОЛЬЗОВАТЕЛЕМ, — СОПРЯЖЕНИЕ УСТРОЙСТВ ПРИ УЧАСТИИ ЧЕЛОВЕКА

Люди помогают машинам аутентифицировать друг друга каждый день — **каждый день**! Вы делаете это, сопрягая беспроводные наушники с телефоном, телефон — с автомобилем, подключая какое-то устройство к домашнему Wi-Fi-роутеру и т. д. В основе любого сопряжения, скорее всего, лежит обмен ключами.

Протоколы аутентификации, описанные в предыдущем разделе, используются в рамках уже защищенных (возможно, с помощью TLS) соединений,

предполагающих аутентификацию сервера. Большая часть этого раздела, напротив, посвящена вопросу создания безопасного канала связи для двух устройств, которые не знают, как аутентифицировать друг друга. Таким образом, речь пойдет о том, как люди могут помочь *преобразовать* небезопасное соединение в *соединение с взаимной аутентификацией*. В связи с этим методы, о которых вы узнаете далее, скорее всего, напомнят вам о некоторых способах установления доверия в рамках протоколов сквозного шифрования, описанных в главе 10, за исключением того, что в данном случае мы будем говорить не о двух людях, пытающихся подтвердить свою личность, а о двух устройствах.

В настоящее время наиболее распространенными способами создания небезопасных соединений, не задействующими Интернет, являются протоколы, основанные на радиочастотах, используемых устройствами малого радиуса действия, такие как Bluetooth, Wi-Fi и NFC (Near Field Communication — связь на малом расстоянии). Технология NFC применяется для платежей с помощью бесконтактной банковской карты или телефона. Спектр устройств, пользующихся этими протоколами связи, как правило, варьируется от маломощной электроники до полнофункциональных компьютеров, что накладывает некоторые ограничения.

- *Устройство, к которому вы пытаетесь подключиться, может не иметь экрана для отображения ключа или способа его ручного ввода.* Мы называем это *приспособленностью* (provisioning) устройства. Например, большинство современных беспроводных аудиогарнитур имеют всего несколько кнопок и больше ничего.
- *Поскольку в процессе проверки участвует человек, необходимость вводить или сравнивать длинные строки часто считается непрактичной и неудобной для пользователя.* По этой причине многие протоколы сокращают связанные с обеспечением безопасности строки до четырех- или шестизначных PIN-кодов.

УПРАЖНЕНИЕ

Представьте себе протокол, в рамках которого требуется ввести правильный четырехзначный PIN-код для безопасного подключения к устройству. Какова вероятность выбрать правильный PIN-код простым угадыванием?

Вероятно, сейчас вы припоминаете свои попытки сопряжения устройств и понимаете, что многие из них *просто сработали*. Например, так.

1. Вы нажали кнопку на устройстве.
2. Устройство переключилось в режим сопряжения.
3. Затем вы отыскали это устройство в списке доступных Bluetooth-устройств на своем телефоне.
4. После нажатия на значок этого устройства произошло его успешное сопряжение с телефоном.

Если вы читали главу 10, то это должно напомнить вам о *доверии при первом использовании* (TOFU). Только на этот раз в дело вступают еще два аспекта:

- *близость* — оба устройства должны находиться рядом друг с другом, особенно при использовании протокола NFC;
- *время* — сопряжение устройств часто предусматривает временное ограничение. Обычно, если в течение 30-секундного интервала сопряжение не состоялось, процесс приходится перезапускать вручную.

Однако, в отличие от подхода TOFU, эти сценарии обычно не позволяют проверить, к тому ли устройству вы подключились. Этот вариант неидеален, поэтому по возможности следует стремиться к обеспечению большей безопасности.

ПРИМЕЧАНИЕ

Кстати, в спецификации ядра Bluetooth TOFU-подобный протокол обозначается фразой Just works («Просто работает»). Следует отметить, что все встроенные протоколы безопасности Bluetooth в настоящее время считаются взломанными из-за множества атак, включая последнюю атаку KNOB, запущенную в 2019 году (https://knobattack.com). Тем не менее техники, описанные в этой главе, безопасны при условии их корректной реализации.

Что еще содержится в нашем наборе инструментов? Далее рассмотрим способы, с помощью которых человек может помочь устройствам пройти аутентификацию.

- Вы увидите, что использование криптографических ключей — всегда самый безопасный, но не всегда самый удобный подход.
- Узнаете о симметричных PAKE-протоколах и о том, как можно ввести один и тот же пароль на двух устройствах для их безопасного соединения.
- Вы познакомитесь с протоколами, основанными на использовании коротких аутентификационных строк (SAS), которые позволяют аутентифицировать результат обмена ключами путем сравнения двух коротких строк, отображаемых двумя устройствами.

Итак, начнем!

11.3.1. Предварительно распределенные ключи

Наивный способ подключения двух устройств друг к другу предполагает использование протоколов, о которых мы говорили в главах 9 и 10 (например, TLS или Noise), и предоставление обоим устройствам общего симметричного секрета или, еще лучше, долгосрочных открытых ключей для обеспечения прямой секретности будущих сеансов. Это означает, что для того, чтобы каждое из устройств могло узнать открытый ключ другого устройства, необходимо:

- предусмотреть способ *экспортировать* открытый ключ с устройства;
- сделать так, чтобы устройство могло *импортировать* открытые ключи.

Как мы увидим далее, этот подход не всегда оказывается простым и удобным. Однако не стоит забывать, что в данном процессе участвует человек, который может наблюдать и, возможно, взаимодействовать с обоими устройствами. В этом заключается отличие от сценариев, которые мы рассматривали раньше, и мы можем использовать это в своих интересах!

> Проблема аутентификации. Одним из основных вопросов криптографии является установление безопасной одноранговой или групповой связи по незащищенному каналу. При отсутствии каких-либо допущений, например о наличии дополнительного безопасного канала, эта задача невыполнима. Однако при наличии некоторых допущений существует множество способов наладить безопасную связь.
>
> *Сильвен Пасини (Secure Communication Using Authenticated Channels, 2009)*

Все описанные далее протоколы основаны на предположении о том, что у вас (человека, отвечающего за весь процесс) есть дополнительный *внеполосный* канал связи, позволяющий безопасно передавать некоторую информацию. Добавление этого внеполосного канала можно смоделировать в виде наличия у двух устройств доступа к каналам двух типов (рис. 11.13):

- к *небезопасному каналу*, например к соединению по Bluetooth или Wi-Fi. По умолчанию пользователь не имеет возможности аутентифицировать устройство, а значит, его может атаковать посредник;
- к *аутентифицированному каналу*, например к экрану устройства. Этот канал обеспечивает целостность/аутентичность передаваемой информации, но не обеспечивает конфиденциальность, поскольку посторонний может заглядывать вам через плечо.



Рис. 11.13. Протоколы аутентификации двух устройств при участии пользователя исходят из того, что эти два устройства задействуют каналы двух типов: небезопасный канал (например, NFC, Bluetooth, Wi-Fi и т. д.), который мы рассматриваем как контролируемый противником, и аутентифицированный канал (например, реальная жизнь), который не обеспечивает конфиденциальность, но может использоваться для обмена относительно небольшими объемами информации

Поскольку этот внеполосный канал обеспечивает низкую степень конфиденциальности, мы обычно используем его не для экспорта секретов, а для передачи

открытых данных. Например, на экран устройства можно вывести открытый ключ или какой-то дайджест. Однако после экспорта открытого ключа вам еще необходимо сделать так, чтобы другое устройство его импортировало. Например, если ключ представляет собой QR-код, то другое устройство должно иметь возможность его отсканировать, а если ключ закодирован в человекочитаемом формате, то пользователь может вручную ввести его на другом устройстве с помощью клавиатуры. После того как оба устройства будут снабжены открытыми ключами друг друга, вы сможете задействовать любые протоколы, о которых мы говорили в главе 9, для взаимно аутентифицированного обмена ключами.

Запомните, что использование криптографических ключей в рамках своего протокола — всегда самый безопасный способ добиться цели, хоть и не всегда самый удобный. Тем не менее реальная криптография полна компромиссов, и именно поэтому две описанные далее схемы не просто существуют, но и являются самыми популярными способами аутентификации устройств.

Давайте посмотрим, как можно применять *пароли* для выполнения взаимно аутентифицированного обмена ключами в тех случаях, когда экспорт и импорт длинных открытых ключей невозможен. Затем мы поговорим о том, как короткие *аутентификационные строки* могут помочь в ситуациях, когда мы не можем импортировать данные на одно или оба устройства.

11.3.2. Симметричный обмен ключами с парольной аутентификацией с протоколом CPace

По возможности следует использовать описанное ранее решение, поскольку оно опирается на надежные асимметричные ключи, выступающие в качестве корня доверия. Правда, на практике ручной ввод длинной строки, представляющей собой ключ, с помощью неудобной клавиатуры оказывается очень утомительным. Пароли гораздо короче, а значит, с ними проще работать. Мы любим пароли, верно? Ну, может быть, мы — нет, но их любят пользователи, а реальная криптография полна компромиссов. Поэтому пусть будет так.

В разделе, посвященном асимметричному обмену ключами с парольной аутентификацией, я упомянул о существовании симметричной (или сбалансированной) версии протокола, в рамках которой два участника, знающие общий пароль, могут выполнить взаимно аутентифицированный обмен ключами. Это именно то, что нам нужно.

Протокол *Composable Password Authenticated Connection Establishment* (CPace) был предложен в 2008 году Бьорном Хаазе и Бенуа Лабриком, а в начале 2020 года выбран исследовательской группой CFRG в качестве официальной рекомендации. Сейчас данный алгоритм стандартизирован в виде RFC. В упрощенном виде протокол работает следующим образом (рис. 11.14).

- Два устройства формируют генератор для некоторой заранее определенной циклической группы на основе общего пароля.
- Затем оба устройства используют этот генератор для обмена эфемерными DH-ключами.

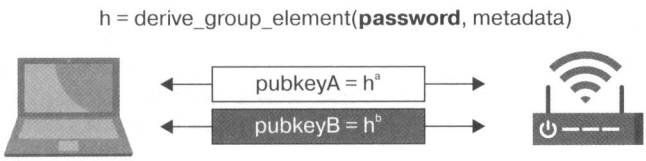


Рис. 11.14. Принцип работы PAKE-протокола CPace заключается в том, что два устройства создают генератор на основе пароля, а затем с его помощью выполняют обычный обмен эфемерными DH-ключами

Дьявол, разумеется, кроется в деталях, и, будучи современной спецификацией, протокол CPace учитывает нюансы эллиптических кривых и определяет, когда необходимо проверять принадлежность полученной точки правильной группе (это связано с использованием модной ныне кривой Curve25519, которая, к сожалению, не охватывает группу простого порядка). Он также определяет способ формирования генератора на основе пароля в группе точек эллиптической кривой с помощью алгоритмов hash-to-curve (хеширование в точку на кривой) и то, как это можно сделать, используя не только общий пароль, но и уникальный идентификатор сеанса и некоторые дополнительные контекстные метаданные, такие как IP-адреса пиров и т. д.

Эти шаги очень важны, поскольку оба участника должны сформировать генератор h таким способом, который не позволит им узнать его дискретный логарифм x, такой, что $g^x = h$. Наконец, сеансовый ключ формируется на основе результата обмена DH-ключами, транскрипта (эфемерных открытых ключей) и уникального идентификатора сеанса.

Интуитивно понятно, что, выдавая себя за одного из участников и посылая групповой элемент в рамках рукопожатия, вы отправляете открытый ключ, связанный с закрытым ключом, которого знать не можете. Это означает, что вы никогда не сможете выполнить обмен DH-ключами, если не знаете пароля. Транскрипт выглядит как обычный обмен DH-ключами, так что у злоумышленника нет шансов, пока алгоритм DH остается безопасным.

11.3.3. Чтобы проверить, не подвергся ли ваш обмен ключами атаке посредника, используйте короткую аутентификационную строку

Во второй части главы вы познакомились с различными протоколами, которые позволяют выполнить сопряжение двух устройств при участии человека. Однако я упомянул о том, что возможности некоторых устройств могут быть настолько ограниченными, что они не смогут задействовать эти протоколы. Давайте рассмотрим схему, которая задействуется в тех случаях, когда два устройства не способны импортировать ключи, но могут сообщить пользователю ограниченное количество данных, например, с помощью экрана, светодиодов, звуковых сигналов и т. д.

В главе 10 вы узнали о способе аутентификации сеанса *после рукопожатия* (то есть обмена ключами) с помощью *отпечатков пальцев* (хешей транскрипта). Мы можем использовать что-то подобное, поскольку у нас есть внеполосный

канал для передачи этих отпечатков. В случае совпадения отпечатков, полученных от обоих устройств, пользователь может быть уверен в том, что обмен ключами не был атакован посредником.

Проблема этих отпечатков заключается в том, что они представляют собой длинные байтовые строки (обычно по 32 байта), которые может быть трудно сообщить пользователю. Кроме того, их сложно сравнивать. Однако для сопряжения устройств мы можем применять гораздо более короткие байтовые строки, поскольку их сравнение выполняется в реальном времени! Такие строки называются *короткими аутентификационными строками* (SAS). В силу удобства для пользователя SAS применяются довольно широко, в частности, в технологии Bluetooth (рис. 11.15).

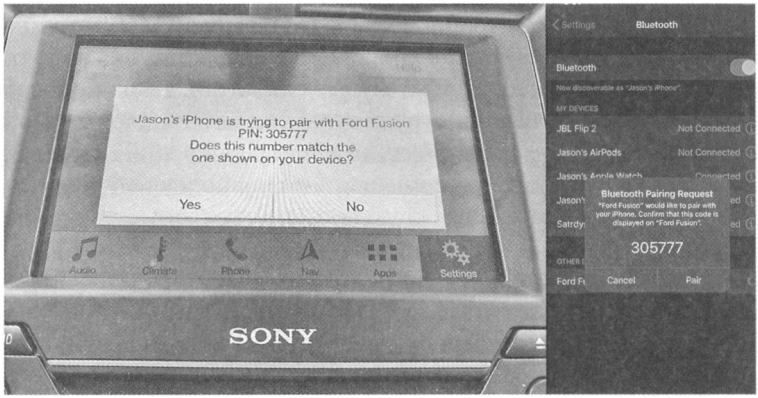


Рис. 11.15. Для сопряжения телефона с автомобилем по Bluetooth можно использовать режим числового сравнения для генерации SAS, согласованной между двумя устройствами. К сожалению, как уже было сказано, из-за атаки KNOB по состоянию на 2021 год протоколы безопасности Bluetooth считаются взломанными. Если вы контролируете оба устройства, вам необходимо реализовать собственный SAS-протокол

Для схем, основанных на использовании SAS, не существует особых стандартов, однако большинство протоколов, включая числовое сравнение в Bluetooth, реализуют тот или иной вариант алгоритма *Диффи — Хеллмана с ручной аутентификацией* (Manually Authenticated Diffie — Hellman, MA-DH). MA-DH представляет собой простой обмен ключами, дополненный трюком, который затрудняет активную реализацию атаки посредника. Вы можете спросить, почему бы просто не создать SAS путем усечения отпечатка? Зачем использовать этот трюк?

Как правило, SAS представляет собой шестизначное число, которое можно получить усечением хеша транскрипта до менее чем 20 бит и его преобразованием в десятичное число. Таким образом, длина SAS оказывается опасно малой, что значительно облегчает злоумышленнику задачу нахождения *второго прообраза* по усеченному хешу. На рис. 11.16 показан пример с двумя устройствами (обозначены именами Алисы и Боба), выполняющими неаутентифицированный обмен ключами. Злоумышленник, активно реализующий атаку посредника, может подменить открытый ключ Алисы, передаваемый в первом сообщении, собственным открытым ключом. После получения открытого ключа Боба он узнает, какое значение

SAS вычислит Боб (усеченный хеш на основе открытого ключа злоумышленника и открытого ключа Боба). Атакующему достаточно сгенерировать множество открытых ключей, чтобы найти такой (`public_keyE2`), при котором SAS Алисы совпадет с SAS Боба.

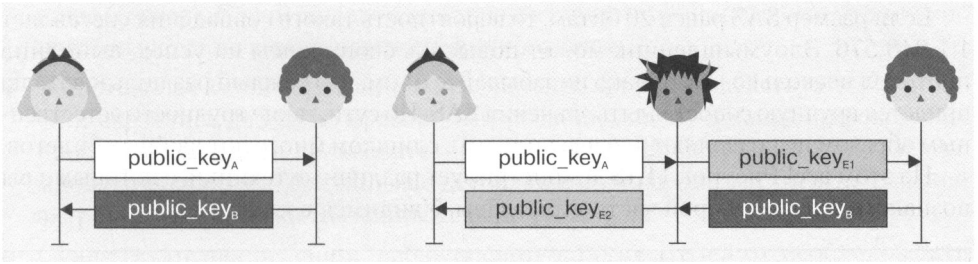


Рис. 11.16. Типичный неаутентифицированный обмен ключами *(слева)* может быть перехвачен злоумышленником, активно реализующим атаку посредника *(справа)*, который сможет подменить открытые ключи Алисы и Боба. Атака окажется успешной, если Алиса и Боб сгенерируют одну и ту же короткую аутентификационную строку, то есть если hash(public_key$_A$ || public_key$_{E2}$) и hash(public_key$_{E2}$ || public_key$_B$) совпадут

На самом деле сгенерировать такой открытый ключ, при котором обе SAS будут совпадать, довольно просто. Допустим, размер SAS составляет 20 бит. В этом случае вам потребуется выполнить всего 2^{20} вычислений, чтобы найти второй прообраз, при котором Алиса и Боб сгенерируют одинаковые SAS. Даже с помощью дешевого телефона эти вычисления можно выполнить довольно быстро.

Главная хитрость обмена ключами на основе SAS заключается в том, чтобы помешать злоумышленнику выбрать второй открытый ключ, обеспечивающий совпадение обеих SAS. Для этого Алиса просто отправляет *обязательство*, фиксирующее значение ее открытого ключа, до того, как увидит открытый ключ Боба (рис. 11.17).

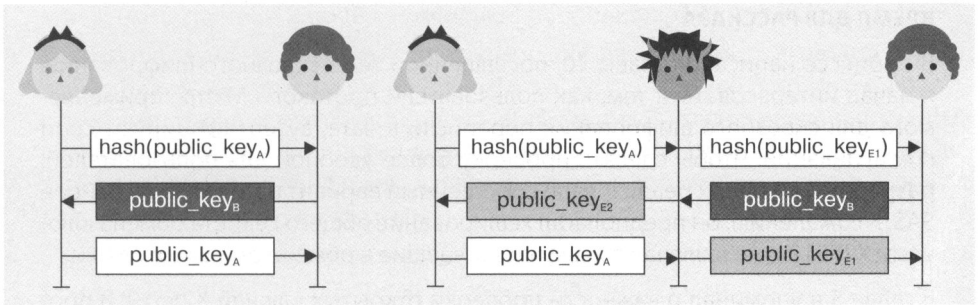


Рис. 11.17. *Слева* — безопасный протокол на основе SAS, в рамках которого Алиса сначала отправляет обязательство, фиксирующее значение ее открытого ключа, и раскрывает этот ключ только после получения открытого ключа Боба. Поскольку Алиса связана обязательством, она не может произвольно выбрать пару своих ключей на основе ключа Боба. *Справа* — в случае активной атаки посредника злоумышленник не сможет обеспечить совпадение SAS Алисы и Боба, выбрав одну из пар ключей

Как и в предыдущей небезопасной схеме, выбор злоумышленником открытого ключа `public_keyE1` не дает ему никаких преимуществ. Однако теперь он не может и выбрать открытый ключ `public_keyE2`, который мог бы ему помочь, поскольку на данном этапе выполнения протокола ему неизвестна SAS Боба. Он вынужден стрелять наугад и надеяться на то, что SAS Алисы и Боба совпадут.

Если размер SAS равен 20 битам, то вероятность такого совпадения составляет 1:1 048 576. Злоумышленник может повысить свои шансы на успех, выполнив протокол несколько раз, однако не забывайте о том, что каждый раз пользователю придется вручную сопоставлять значения SAS. По сути, такие трудности естественным образом не позволяют злодею получить слишком много лотерейных билетов.

На этом все! Рисунок 11.18 демонстрирует различные техники, с которыми вы познакомились во второй части этой главы. Увидимся в главе 12.

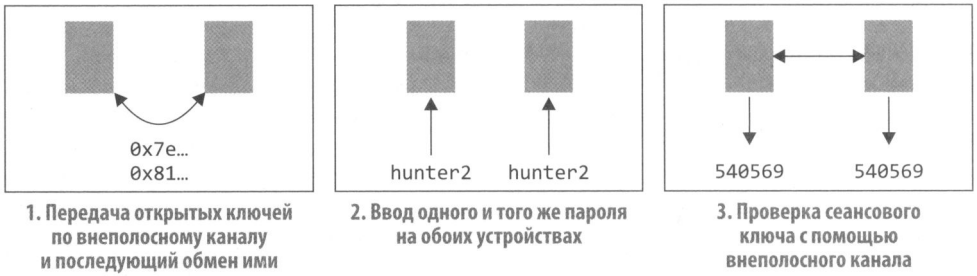


Рис. 11.18. Вы узнали о трех методах сопряжения двух устройств: пользователь помогает устройствам получить открытые ключи друг друга, чтобы они могли выполнить обмен ключами *(1)*; пользователь вводит один и тот же пароль на двух устройствах, чтобы они смогли выполнить симметричный обмен ключами с парольной аутентификацией *(2)*; пользователь проверяет отпечаток после обмена ключами, чтобы убедиться в том, что процесс сопряжения не подвергся атаке посредника *(3)*

ВРЕМЯ ДЛЯ РАССКАЗА

В процессе написания главы 10, посвященной теме сквозного шифрования, я начал интересоваться тем, как пользователи протокола Matrix, применяемого для сквозного шифрования переписки в чате, аутентифицируют свои коммуникации. Чтобы сделать проверку более удобной для пользователей, разработчики Matrix реализовали собственный вариант протокола на основе SAS. К сожалению, он предполагал хеширование общего секрета обмена ключами X25519 и не включал в хеш участвовавшие в обмене открытые ключи.

В главе 5 я упоминал о важности проверки открытых ключей X25519. В протоколе Matrix она не была предусмотрена, что позволило злоумышленнику отправить пользователям неверные открытые ключи, из-за чего они в итоге получили одинаковый предсказуемый общий секрет и, в свою очередь, одно и то же значение SAS. Это полностью перечеркнуло обещанное сквозное шифрование, но было быстро исправлено разработчиками Matrix.

РЕЗЮМЕ

- Протоколы аутентификации пользователей (то есть протоколы, позволяющие машинам аутентифицировать людей) часто выполняются через защищенные соединения, при установлении которых аутентификацию проходит только машина (сервер). В этом смысле одностороннее аутентифицированное соединение преобразуется во взаимно аутентифицированное.
- В протоколах аутентификации пользователей активно задействуются пароли. Они оказались довольно практичным решением, получившим широкое распространение среди пользователей. Однако это привело к возникновению множества проблем, связанных с несоблюдением парольной гигиены, низкой энтропией и утечками баз данных с паролями.
- Существуют два решения, избавляющие пользователей от необходимости запоминать множество паролей (и предотвращающих их многократное применение).
  - *Менеджеры паролей* — инструменты, с помощью которых пользователи могут генерировать надежные пароли для каждого используемого приложения и управлять ими.
  - *Технология единого входа* (SSO) — федеративный протокол, позволяющий пользователю применять одну учетную запись для регистрации и авторизации в других сервисах.
- Чтобы серверы не узнавали и не хранили пароли своих пользователей, можно выполнять асимметричный обмен ключами с парольной аутентификацией, то есть задействовать асимметричный PAKE-протокол. Такой протокол, например OPAQUE, позволяет пользователям аутентифицироваться на сервере с помощью паролей, не раскрывая ему сами пароли.
- Чтобы полностью отказаться от паролей, пользователи могут применять симметричные ключи с помощью алгоритмов создания одноразовых паролей (OTP) или асимметричные ключи с помощью таких стандартов, как FIDO2.
- Протоколы аутентификации при участии пользователя часто выполняются по незащищенным каналам (Wi-Fi, Bluetooth, NFC) и позволяют двум устройствам аутентифицировать друг друга. Для защиты каналов связи в подобных сценариях эти протоколы предполагают наличие дополнительного аутентифицированного (но не конфиденциального) канала, который могут использовать устройства (например, экран).
- Экспорт открытого ключа с одного устройства на другое мог бы обеспечить взаимно аутентифицированный обмен ключами. К сожалению, соответствующие потоки не очень удобны для пользователей, а иногда и невозможны из-за ограничений устройства (например, при отсутствии возможности экспортировать или импортировать ключи).
- Протоколы симметричного обмена ключами с парольной аутентификацией (симметричные PAKE-протоколы), такие как CPace, позволяют снизить нагрузку на пользователя, связанную с импортом длинного открытого ключа, сводя

ее к ручному вводу пароля на устройстве. Симметричные PAKE-протоколы уже применяются большинством людей, например, при подключении устройств к домашнему Wi-Fi-роутеру.

- Протоколы на основе коротких аутентификационных строк (SAS) могут обеспечить безопасность устройств, которые не способны импортировать ключи или пароли, но способны отобразить короткую строку после обмена ключами. Эта короткая строка должна быть одинаковой на обоих устройствах, чтобы гарантировать: неаутентифицированный обмен ключами не подвергся активной атаке посредника.

12 Криптография и криптовалюты

В этой главе

- ✓ Протоколы консенсуса и их роль в существовании криптовалют.
- ✓ Различные виды криптовалют.
- ✓ Принцип работы криптовалют Bitcoin и Diem.

Способна ли криптография стать основой новой финансовой системы? На этот вопрос криптовалюты пытаются ответить как минимум с 2008 года, когда появился биткойн, разработанный Сатоши Накамото, истинная личность которого до сих пор остается секретом. До этого термин «крипто» применялся только к области криптографии. Однако после создания биткойна его значение быстро изменилось, и теперь он используется и применительно к криптовалютам. Криптоэнтузиасты, в свою очередь, стали более активно интересоваться криптографией. Это вполне логично, если учесть, что она лежит в основе криптовалют.

Что же такое *криптовалюта*?

- *Во-первых, это цифровая валюта*, которая позволяет людям совершать денежные операции в электронном виде. Иногда при этом используется валюта, обеспеченная государством, например доллар США, а иногда — выдуманная, например биткойн. Вы, скорее всего, уже имеете дело с цифровыми валютами, например, когда отправляете кому-то деньги через Интернет или пользуетесь расчетным счетом. Действительно, людям больше не нужно пересылать наличные по почте, а большинство денежных операций сегодня сводятся к простому обновлению значений в строках баз данных.

- *Во-вторых, это валюта, которая в значительной степени опирается на криптографию, что позволяет исключить участие доверенной третьей стороны и обеспечить прозрачность.* Криптовалюта не предполагает наличия центрального органа вроде правительства или банка, которому приходится слепо доверять. Обычно мы называем это свойство *децентрализацией* (например, когда говорим о децентрализации доверия).

Криптовалюты появились сравнительно недавно. Первым успешным экспериментом стал биткойн, предложенный в 2008 году, в разгар мирового финансового кризиса. Хотя этот кризис начался в США, он быстро распространился на другие страны, подорвав доверие людей к финансовым системам и подготовив почву для реализации более прозрачных инициатив, таких как биткойн. В то время многие люди начали понимать, что существовавший на тот момент способ проведения финансовых операций был неэффективным, дорогостоящим и непрозрачным для большинства пользователей. Остальное — уже история, и я считаю, что эта книга является первой книгой по криптографии, включающей в себя главу, посвященную криптовалютам.

12.1. ЗНАКОМСТВО С ВИЗАНТИЙСКИМИ ОТКАЗОУСТОЙЧИВЫМИ АЛГОРИТМАМИ КОНСЕНСУСА

Представьте, что вы хотите создать новую цифровую валюту. На самом деле изобрести нечто работающее не так уж сложно. Можно создать базу данных на выделенном сервере, которая будет применяться для отслеживания пользователей и их балансов. С ее помощью вы предоставите людям интерфейс, позволяющий проверять свой баланс или совершать платежи, в результате чего их баланс в базе данных будет уменьшаться, а баланс других пользователей — увеличиваться. Изначально вы также можете случайным образом раздать своим друзьям некоторое количество придуманной вами валюты, чтобы они могли начать переводить деньги в вашу систему. Однако такая простая схема имеет ряд недостатков.

12.1.1. Проблема устойчивости и распределенные протоколы

Описанная система имеет *единую точку отказа*. Если у вас отключат электричество, то клиенты не смогут пользоваться системой. Хуже того, если ваш сервер будет уничтожен в результате стихийного бедствия, все пользователи лишатся своих средств. Для решения этой проблемы существуют известные методы, которые вы можете использовать для повышения устойчивости своей системы. Изучением этих методов занимается область *распределенных систем*.

В нашем случае самым очевидным решением, применяемым в большинстве крупных приложений, является репликация (то есть копирование содержимого базы данных на резервные серверы), которая происходит практически в режиме реального времени. Эти серверы могут располагаться в различных географических

точках и использоваться для восстановления данных и даже выступать в качестве основного сервера, если тот выйдет из строя. Теперь у вас есть *распределенная база данных* с *высокой доступностью*.

В больших системах, обслуживающих множество запросов, часто бывает так, что резервные базы данных не просто лежат в сторонке, ожидая, когда в них возникнет необходимость, а используются для чтения состояния. Выполнять обновления и делать записи сразу в нескольких базах данных довольно трудно и чревато конфликтами (как в тех случаях, когда два человека редактируют один и тот же документ). Поэтому часто требуется сделать так, чтобы одна база данных отвечала за все записи и обновления, выступая в роли *лидера*, а остальные применялись для чтения состояния.

Репликация содержимого базы данных может требовать много времени, и, вполне вероятно, некоторые из ваших баз данных будут отставать от лидера. Это особенно верно, если они расположены на большом расстоянии друг от друга или по каким-то причинам сталкиваются с сетевыми задержками. Это отставание становится проблемой, когда реплицированные базы данных используются для чтения состояния. (Представьте, что вы с другом видите разные балансы счета из-за того, что посылаете запрос на разные серверы.)

Для того чтобы справиться с этой проблемой, пишут специальные приложения. Этот подход называется *согласованностью в конечном счете*, поскольку в итоге состояния баз данных становятся одинаковыми. (Существуют и более сильные модели согласованности, но обычно они довольно медленные и непрактичные.) Однако у таких систем есть и другие проблемы. Например, какая из резервных баз данных заменит основную, если та выйдет из строя? И если в момент сбоя резервные базы данных отставали от основной, не потеряем ли мы часть последних изменений?

Когда вам нужно обеспечить согласие (то есть консенсус) всей системы по поводу какого-то решения, можете использовать более надежные *алгоритмы консенсуса*, также называемые алгоритмами *репликации журнала*, *репликации конечного автомата* или *атомарной рассылки*. Представьте, что алгоритм консенсуса решает проблему, с которой сталкивается группа людей, пытающихся договориться о том, какую пиццу им стоит заказать. Если все они находятся в одной комнате, выяснить предпочтения большинства не составит труда. Но если общаются через сеть, в которой сообщения могут задерживаться, обрываться, перехватываться и изменяться, то им потребуется более сложный протокол.



Давайте посмотрим на то, как консенсус помогает ответить на два предыдущих вопроса. Первый вопрос о том, какая из резервных баз данных заменит основную, если та выйдет из строя, решается с помощью алгоритма консенсуса, называемого алгоритмом *выбора лидера*. Второй вопрос часто решается путем учета изменений в базе данных на двух разных этапах: *ожидания* и *фиксации*. Сначала изменения состояния базы данных находятся в режиме ожидания и фиксируются только тогда, когда достаточное количество баз данных достигает согласия (здесь также может пригодиться протокол консенсуса). После фиксации обновление состояния, скорее всего, не будет утрачено, поскольку большинство участвующих в процессе баз данных зафиксировали соответствующее изменение.

К хорошо известным алгоритмам консенсуса относятся Paxos (предложенный Лэмпортом в 1989 году) и его упрощенная версия Raft (предложенная Онгаро и Оустерхаутом в 2013 году). Эти алгоритмы можно использовать в большинстве распределенных систем баз данных для решения различных задач. (Отличное интерактивное объяснение алгоритма Raft доступно по адресу https://thesecretlivesofdata.com/raft.)

12.1.2. Децентрализация помогает решить проблему доверия

В плане эксплуатации распределенные системы представляют собой устойчивую альтернативу тем системам, которые выступают в качестве единой точки отказа. Алгоритмы консенсуса, используемые в большинстве распределенных систем баз данных, плохо переносят сбои. Как только машины начинают сбоить, вести себя неправильно из-за аппаратных неисправностей или терять связь с другими машинами, например с целыми частями сети, возникают проблемы. Более того, пользователь никак не может выявить эти сбои, что становится еще более серьезной проблемой в случае взлома серверов.

Если я отправляю серверу запрос и он сообщает мне, что у Алисы на счету 5 млрд долларов, мне приходится просто ему верить. Если же сервер включит в свой ответ данные обо всех денежных переводах, полученных и отправленных Алисой за все время, и просуммирует их, я смогу убедиться в том, что баланс ее счета действительно составляет 5 млрд долларов. Но как я могу удостовериться в том, что сервер мне не солгал? Возможно, когда Боб обращается к другому серверу, тот сообщает ему совершенно другой баланс и/или историю счета Алисы. Представление двух противоречащих друг другу состояний в качестве действительных называется *развилкой* (fork), разветвлением истории, которого не должно было произойти. Вероятно, вы уже догадались, что компрометация одной из реплицированных баз данных может привести к весьма разрушительным последствиям.

В главе 9 я упомянул о *Certificate Transparency* (прозрачность сертификатов) — протоколе-сплетнике, который направлен на обнаружение подобных развилок в веб-инфраструктуре открытых ключей. Однако, когда речь идет о деньгах, простого обнаружения развилок недостаточно. Нам необходимо предотвратить их возникновение! В 1982 году Лэмпорт, автор алгоритма консенсуса Paxos, представил идею *византийских отказоустойчивых* (BFT) *алгоритмов консенсуса*.

> Представим, что несколько дивизий византийской армии расположились лагерем у вражеского города. Каждой из дивизий командует свой генерал. Эти генералы могут общаться друг с другом только через гонца. В ходе наблюдения за противником они должны выработать общий план действий. Однако некоторые генералы могут оказаться предателями и попытаться помешать верным генералам прийти к согласию.
>
> *Лэмпорт и др. (The Byzantine Generals Problem, 1982)*

Своей аналогией Лэмпорт положил начало развитию области алгоритмов консенсуса BFT (Byzantine fault-tolerance — византийская отказоустойчивость), направленных на то, чтобы не позволить злонамеренным участникам создавать противоречивые представления о системе при согласовании решения. BFT-алгоритмы очень похожи на такие алгоритмы консенсуса, как Paxos и Raft, за исключением того, что реплицированные базы данных (участники протокола) больше не доверяют друг другу слепо. BFT-протоколы обычно активно применяют криптографию для аутентификации сообщений и решений, которые, в свою очередь, могут быть использованы другими участниками для криптографической проверки решений, принятых протоколом консенсуса.

Таким образом, BFT-протоколы позволяют решить проблемы устойчивости и доверия. Различные реплицированные базы данных могут задействовать BFT-алгоритмы для согласования новых состояний системы (например, балансов пользователей) и при этом контролировать друг друга, удостоверяясь в том, что переходы состояний (транзакции между пользователями) действительны и согласованы большинством участников. В таких случаях мы говорим о *децентрализации* доверия.

Первым реальным BFT-алгоритмом стал *Practical BFT* (PBFT), опубликованный в 1999 году. Подобно алгоритмам Paxos и Raft, PBFT предполагает выбор лидера, вносящего предложения, а остальные участники пытаются прийти к соглашению по поводу них. К сожалению, алгоритм PBFT довольно сложен, медлителен и плохо масштабируется по мере увеличения количества участников. Большинство современных криптовалют используют более эффективные варианты алгоритма PBFT. Например, криптовалюта Diem, представленная компанией Facebook в 2019 году, основана на протоколе HotStuff, вдохновленном PBFT.

12.1.3. Проблема масштаба. Общедоступные и устойчивые к цензуре сети

Одним из ограничений алгоритмов консенсуса на основе PBFT является то, что все они предполагают известное и фиксированное количество участников. Если определенное число участников превышено, они начинают разрушаться: резко возрастает их коммуникационная сложность, работа сильно замедляется, процесс выбора лидера становится слишком затруднительным и т. д.

Как же криптовалюта определяет участников консенсуса? Для этого существуют несколько механизмов, но наиболее широко распространены следующие два:

- *Proof of authority* (PoA, доказательство полномочий) — участники консенсуса определяются заранее;

- *Proof of stake* (PoS, доказательство доли владения) — участники консенсуса выбираются динамически, исходя из того, кто поставил на кон наибольшую сумму и, следовательно, меньше заинтересован в атаках на протокол. Как правило, системы, основанные на механизме PoS, выбирают участников в зависимости от количества цифровой валюты, которой те владеют.

Однако не все протоколы консенсуса являются классическими BFT-протоколами. Биткойн, например, использует механизм консенсуса, который не предполагает наличия известного списка участников. В свое время эта идея была довольно новой, и ее реализация потребовала некоторого ослабления ограничений классических BFT-протоколов. Как вы увидите далее в этой главе, благодаря такому подходу биткойн допускает создание форков (ответвлений), что порождает дополнительные проблемы.

Как же выбрать лидера при отсутствии участников? Для этого можно было бы использовать PoS-систему, например протокол консенсуса Ouroboros. Однако вместо этого алгоритм консенсуса системы Bitcoin опирается на вероятностный механизм, называемый *доказательством выполнения работы* (proof of work, PoW). В системе Bitcoin это означает: для того чтобы стать участником и лидером, людям необходимо решать головоломки. И как будет показано далее в этой главе, это криптографические головоломки.

Из-за отсутствия списка известных участников систему Bitcoin называют *общедоступной* сетью. В ней вам не нужны дополнительные разрешения для участия в процессе достижения консенсуса — участвовать может любой желающий. В этом заключается отличие общедоступных от *эксклюзивных* сетей, список участников которых фиксирован. Некоторые из этих новых концепций схематично представлены на рис. 12.1.

До недавнего времени было непонятно, как можно использовать классические BFT-протоколы в общедоступной сети, участником которой может быть любой желающий. Сегодня существует множество основанных на механизме PoS способов динамического выбора подмножества участников процесса достижения консенсуса. Один из самых известных — опубликованный в 2017 году алгоритм Algorand, который динамически выбирает участников и лидеров в зависимости от количества валюты, которой они владеют.

Также считается, что сеть Bitcoin устойчива к цензуре, поскольку вы не можете заранее предугадать, кто станет следующим лидером, а значит, и помешать системе его избрать. Не вполне понятно, возможно ли такое в PoS-системах, где вычислить личность владельца большого количества валюты относительно легко.

Следует отметить, что не все BFT-протоколы предполагают выбор лидера. Некоторые из них *безлидерные*, то есть не требуют того, чтобы избранные лидеры принимали решения относительно последующих переходов состояний. Вместо этого любой желающий может предложить изменение, а протокол консенсуса помогает участникам прийти к согласию по поводу следующего состояния. В 2019 году компания Avalanche запустила криптовалюту, позволяющую всем желающим предлагать изменения и участвовать в достижении консенсуса.

Наконец, если вы подумали, что децентрализованная платежная система не может обойтись без механизма консенсуса, то это не так. Протоколы, не требующие

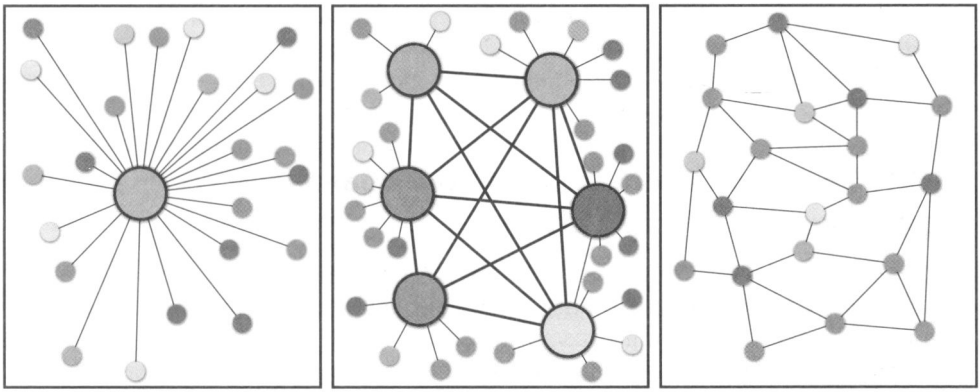


Рис. 12.1. Централизованная сеть может рассматриваться как единая точка отказа, в то время как распределенная и децентрализованная сеть устойчива к отключению нескольких серверов или к их злонамеренным действиям. Эксклюзивная сеть предполагает наличие известного фиксированного списка основных участников, тогда как в общедоступной сети участником может стать любой желающий

консенсуса, были предложены в 2018 году в работе Геррауи, Кузнецова, Монти, Павловича и Середински *AT2: Asynchronous Trustworthy Transfers*. Учитывая то, что это относительно новые протоколы и они еще не прошли полевые испытания, я не буду говорить о них в этой главе. В остальных ее разделах речь пойдет о двух разных криптовалютах, с помощью которых я продемонстрирую различные аспекты этой области.

- *Bitcoin* — самая популярная криптовалюта на основе механизма PoW, появившаяся в 2008 году.
- *Diem* — криптовалюта, основанная на BFT-протоколе, анонсированная Facebook и группой других компаний в 2019-м.

12.2. ПРИНЦИП РАБОТЫ СИСТЕМЫ BITCOIN

Тридцать первого октября 2008 года анонимный исследователь (или группа исследователей), использующий псевдоним Сатоши Накамото, опубликовал документ под названием *Bitcoin: A Peer-to-Peer Electronic Cash System* («Биткойн: система пиринговой электронной наличности»). Истинная личность автора до сих пор остается неизвестной. Вскоре после этого он выпустил официальный клиент Bitcoin — программное обеспечение, которое мог запустить любой желающий, чтобы присоединиться к сети и участвовать в ее работе. Участников, использующих одно и то же программное обеспечение или хотя бы один и тот же алгоритм, оказалось достаточно для того, чтобы появилась первая в истории криптовалюта — биткойн, или BTC.

Биткойн — это настоящая история успеха. Данная криптовалюта существует уже более десяти лет (на момент написания этой книги), позволяя пользователям

всего мира совершать транзакции с помощью цифровых денег. В 2010 году программист Ласло Ханеч купил две пиццы за 10 000 BTC. Сейчас (в феврале 2021-го) один BTC стоит почти 57 000 долларов. Как видите, криптовалюты могут быть чрезвычайно волатильными.

12.2.1. Обработка пользовательских балансов и транзакций в системе Bitcoin

Давайте углубимся во внутреннее устройство системы Bitcoin и рассмотрим, как она обрабатывает пользовательские балансы и транзакции. Будучи пользователем Bitcoin, вы имеете дело с криптографией. Вместо имени пользователя и пароля для входа на сайт, как это бывает в случае с банками, вы имеете пару ключей ECDSA (Elliptic Curve Digital Signature Algorithm), которую генерируете сами. Баланс пользователя представляет собой количество BTC, связанное с открытым ключом, поэтому для получения BTC вы можете просто предоставить свой открытый ключ другим людям.

Чтобы задействовать свои BTC, вы подписываете транзакцию своим закрытым ключом. Опустив некоторые детали, которые я объясню позже, суть транзакции можно выразить простой фразой: «Я отправляю *X* BTC на адрес, соответствующий открытому ключу *Y*».

ПРИМЕЧАНИЕ

В главе 7 я упомянул о том, что система Bitcoin использует кривую secp256k1 с алгоритмом ECDSA. Эту кривую не следует путать с кривой P-256 от NIST, которая известна как secp256r1.

Сохранность ваших средств напрямую зависит от надежности вашего закрытого ключа. А как вы уже знаете, управление ключами — дело непростое. За последнее десятилетие проблемы с управлением ключами от криптовалютных кошельков привели к случайной потере или краже цифровой валюты на миллионы долларов. Так что будьте осторожны!

Система Bitcoin предусматривает различные типы транзакций, и большинство транзакций, наблюдаемых в сети, на самом деле скрывают открытый ключ получателя путем его хеширования. В таких случаях хеш открытого ключа называется *адресом* счета. (Мой биткойн-адрес выглядит так: `bc1q8y6p4x3rp32dz80etpyffh6764ray9842egchy`.) Он скрывает фактический открытый ключ счета до тех пор, пока его владелец не решит потратить свои BTC (в этом случае ему необходимо раскрыть прообраз своего адреса, чтобы другие пользователи могли проверить подпись). Это делает адреса короче и не позволяет посторонним получить ваш секретный ключ в случае нарушения алгоритма ECDSA.

Существование различных типов транзакций — интересная особенность системы Bitcoin. Транзакции — это не просто полезная нагрузка, содержащая определенную информацию, на самом деле они представляют собой короткие сценарии, или скрипты, написанные с помощью довольно ограниченного набора специально созданных инструкций. При обработке транзакции сначала выполняется скрипт,

а затем полученный вывод определяет действительность транзакции и дальнейшие шаги, необходимые для изменения состояния всех счетов.

В таких системах, как Ethereum, идея скриптинга доведена до предела, что выражается в возможности запуска гораздо более сложных программ (так называемых *смарт-контрактов*) при выполнении транзакции. Здесь мне стоит прояснить несколько моментов.

1. Что понимается под транзакцией?
2. Что понимается под выполнением транзакции? И кто именно ее выполняет?

На второй вопрос я отвечу в следующем разделе. А пока давайте разберемся с первым.

Особенность системы Bitcoin заключается в отсутствии реальной базы данных, содержащей сведения об остатках на счетах. Вместо этого у пользователя есть кошельки с BTC, которые он может потратить. Они называются *неизрасходованными транзакционными выходами* (unspent transaction output, UTXO). UTXO можно уподобить большой банке, открытой для всеобщего обозрения и наполненной монетами, потратить которые может только ее владелец. Когда в ходе транзакции часть монет тратится, они исчезают из банки отправителя и появляются в банке получателя. Эти новые монеты представляют собой транзакционный выход.

Чтобы узнать, сколько BTC хранится у вас на счете, нужно посчитать все UTXO, присвоенные вашему адресу. Другими словами, необходимо сосчитать все деньги, которые были отправлены вам и которые вы еще не потратили. Пример, иллюстрирующий использование UTXO в ходе выполнения транзакций, показан на рис. 12.2.

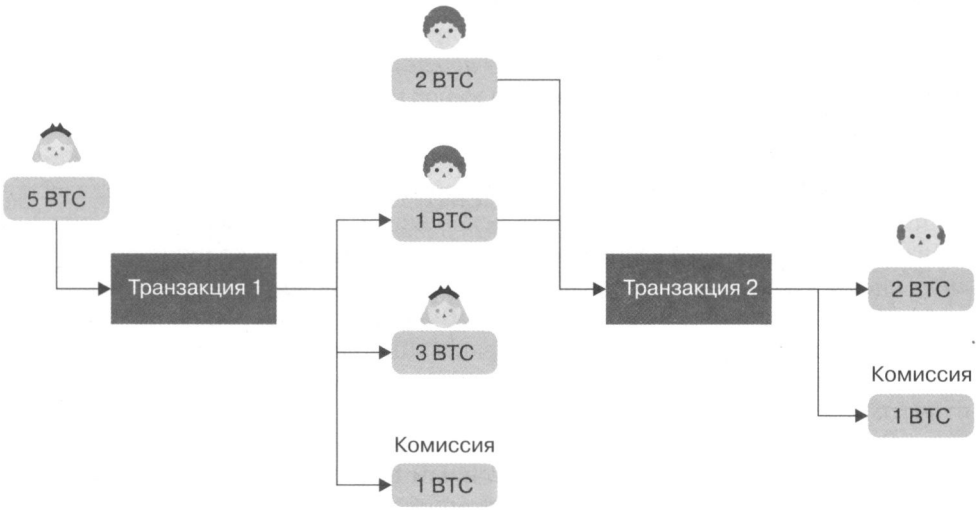


Рис. 12.2. Транзакция 1 подписывается Алисой и предполагает перевод 1 BTC Бобу. Поскольку в ней используется UTXO в 5 BTC, Алисе должна быть отправлена сдача, часть которой должна быть зарезервирована в качестве комиссии. Транзакция 2 подписывается Бобом и объединяет два UTXO для перевода 2 BTC Феликсу (в реальности комиссии гораздо ниже)

Теперь возникает вопрос, напоминающий дилемму курицы и яйца: откуда взялись первые UTXO? На него я отвечу в следующем разделе.

12.2.2. Добыча BTC в эпоху цифрового золота

Теперь вы понимаете, что представляет собой транзакция в системе Bitcoin, и знаете, как можно управлять своим счетом и выяснять баланс других пользователей. Но кто же отслеживает все эти транзакции? Ответ: все участники!

При использовании биткойнов каждая транзакция должна быть опубликована и зафиксирована в истории. Система Bitcoin представляет собой реестр, *в котором возможно только добавление данных*, — бухгалтерскую книгу, где каждая страница связана с предыдущей. Это означает, что вы не можете вернуться назад и изменить страницу. А поскольку каждая транзакция является публичной, вся анонимность, на которую вы можете рассчитывать, сводится к сложности выяснения реальной личности владельца того или иного открытого ключа.

Любую транзакцию, выполненную за время существования биткойна, можно легко проверить, установив клиент Bitcoin и загрузив с его помощью всю историю. При этом вы становитесь частью сети и вынуждены заново проводить каждую транзакцию в соответствии с правилами, закодированными в клиенте Bitcoin. Разумеется, история Bitcoin просто огромна: на момент написания этой книги ее размер составлял около 300 Гбайт, поэтому загрузка может длиться до нескольких дней в зависимости от скорости вашего интернет-соединения. Гораздо легче проверить транзакции, воспользовавшись онлайн-сервисом, который делает всю тяжелую работу за вас (конечно, при условии, что вы ему доверяете). Один из таких *блокчейн-обозревателей* показан на рис. 12.3.

По сути, биткойн представляет собой список всех транзакций, которые были обработаны с момента его создания (*добычи генезис-блока*) до настоящего времени. Вероятно, вам интересно, кто отвечает за выбор и упорядочение транзакций в этой бухгалтерской книге?

Чтобы достичь согласия относительно порядка транзакций, система Bitcoin позволяет любому человеку (даже вам) предложить список транзакций для включения в следующую страницу бухгалтерской книги. Это предложение со списком транзакций называется *блоком*. Однако, учитывая огромное количество участников в системе Bitcoin, предоставление каждому из них возможности предложить блок — это настоящий рецепт катастрофы. Вместо этого нам нужно, чтобы только один человек мог предлагать список транзакций для включения в следующий блок. Для этого система Bitcoin дает всем участникам возможность решить вероятностную головоломку. Тот, кто сделает это первым, сможет предложить свой блок. В этом и заключается суть механизма доказательства выполнения работы (PoW), о котором я говорил ранее. Механизм PoW системы Bitcoin основан на поиске блока, хеш которого меньше некоторого значения. Другими словами, дайджест блока должен иметь двоичное представление, начинающееся с некоторого заданного количества нулей.

Помимо транзакций, которые вы хотите включить, блок должен содержать хеш предыдущего блока. Таким образом, реестр биткойна представляет собой последовательность блоков, каждый из которых ссылается на предыдущий вплоть

до самого первого генезис-блока. Эта последовательность называется *блокчейном* (цепочкой блоков). Прелесть блокчейна заключается в том, что малейшее изменение блока делает всю цепочку недействительной, так как дайджест блока также изменяется и, следовательно, портит ссылку на него, содержащуюся в следующем блоке.



Рис. 12.3. Случайная транзакция, которую я выбрал для анализа на сайте https://blockchain.com (http://mng.bz/n295). Она использует один вход (около 1,976 BTC) и разделяет его на два выхода (примерно по 0,009 BTC и 1,967 BTC). Разница между общей суммой на входе и общей суммой на выходе — это комиссия за транзакцию (не представленная как выход). Остальные поля представляют собой скрипты, написанные на скриптовом языке Bitcoin для того, чтобы потратить UTXO на входе или сделать расходуемыми UTXO на выходе

Обратите внимание на то, что вам, как участнику, желающему предложить следующий блок, не обязательно вносить в него много изменений для получения из него нового хеша. Чтобы повлиять на хеш блока, вы можете сначала исправить большую часть его содержимого (список включенных в него транзакций, хеш предыдущего, родительского блока и т. д.), а затем изменить значение nonce-числа в соответствующем поле. Вы можете рассматривать это поле в качестве счетчика, увеличивая его значение до тех пор, пока не найдете дайджест, соответствующий правилам игры, или сгенерировать случайное значение. Я проиллюстрировал эту идею на рис. 12.4.

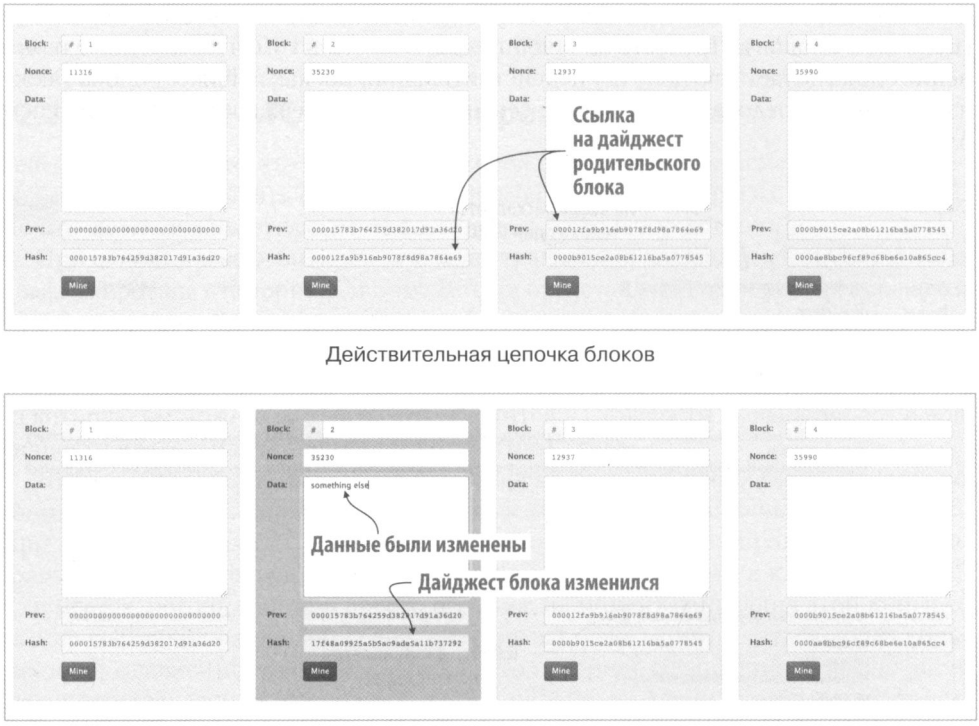


Рис. 12.4. На сайте https://andersbrownworth.com/blockchain/blockchain вы можете в интерактивном режиме поэкспериментировать с игрушечным блокчейном. Каждый блок включает в себя дайджест родительского блока и случайное значение nonce, которое позволяет его дайджесту начинаться с четырех нулей. Обратите внимание на то, что это верно только для верхней цепочки блоков. Нижняя цепочка содержит измененный блок (№ 2), который изначально не содержал никаких данных. Поскольку это изменение привело к изменению его дайджеста, он больше не может быть аутентифицирован последующими блоками

Эта система работает, потому что все ее участники используют один и тот же протокол и подчиняются одним и тем же правилам. Синхронизируясь с блокчейном, вы загружаете блоки остальных пиров и удостоверяетесь в следующем:

- хеширование каждого блока действительно дает дайджест меньше некоторого ожидаемого значения;
- каждый блок ссылается на предыдущий блок в истории.

Не все участники обязаны предлагать блоки, но при желании вы можете это сделать. Предлагая блок, вы становитесь так называемым *майнером*. Это означает, что для того, чтобы ваши транзакции попали в блокчейн, вам нужна помощь майнеров (рис. 12.5).

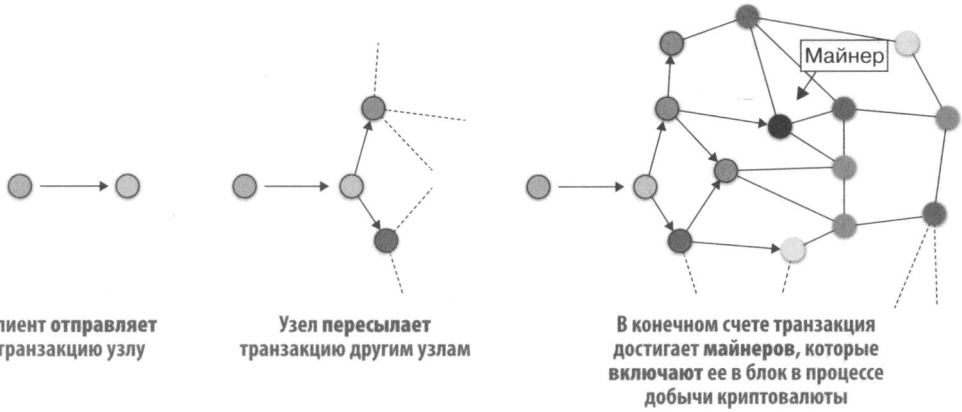


Рис. 12.5. Сеть Bitcoin состоит из множества связанных между собой узлов, некоторыми из них являются майнеры. Чтобы выполнить транзакцию, вы должны отправить ее майнеру, способному поместить ее в блокчейн, то есть включить в блок. Поскольку вы не знаете, какому майнеру удастся добыть очередной блок, то должны распространить свою транзакцию по сети, чтобы она достигла максимально возможного числа майнеров

Майнеры работают не бесплатно. Если майнер находит блок, он получает:

- *вознаграждение* — фиксированное количество BTC, создаваемое и отправляемое на его адрес. В самом начале майнеры получали 50 BTC за добытый блок. Однако через каждые 210 000 блоков размер вознаграждения уменьшается вдвое и в конечном итоге достигнет 0, ограничив общее количество BTC 21 млн;
- *все комиссии за транзакции, содержащиеся в блоке.* Именно поэтому увеличение комиссий за транзакции повышает скорость их выполнения, так как майнеры склонны включать в добываемые блоки транзакции с более высокой комиссией.

Так пользователи системы Bitcoin получают стимул к дальнейшему развитию протокола. Каждый блок содержит адрес, на который поступают вознаграждение и комиссии, называемый *coinbase*. Как правило, это собственный адрес майнера.

Теперь мы можем ответить на вопрос, поставленный в начале раздела: откуда взялись первые UTXO? Ответ заключается в том, что все BTC были в тот или иной момент истории созданы как часть вознаграждения майнеров за добычу блока.

12.2.3. Форки. Разрешение конфликтов при добыче криптовалюты

Система Bitcoin распределяет задачу выбора следующего набора транзакций для обработки с помощью механизма PoW. Ваш шанс добыть блок напрямую зависит от количества хешей, которые вы можете вычислить, то есть от количества расчетов, которые можете выполнить. Сейчас большой объем вычислительных мощностей используется для добычи блоков биткойна и других криптовалют, основанных на механизме PoW.

ПРИМЕЧАНИЕ

Механизм PoW можно рассматривать как способ защиты системы Bitcoin от так называемых *атак Сивиллы*, в основе которых лежит возможность создавать произвольное количество учетных записей, что дает нечестным участникам несправедливое преимущество. В системе Bitcoin увеличить вычислительные мощности можно, только купив дополнительное оборудование для вычисления хешей, а не за счет создания большого количества адресов в сети.

Однако остается одна проблема: задача поиска хеша меньше некоторого значения не должна быть слишком легкой. Если бы это было так, то действительный блок одновременно добывался бы слишком большим количеством участников. В таком случае было бы непонятно, какой из добытых блоков должен стать законным следующим блоком в цепи. Это и называется *форком*.

Для решения проблемы форков система Bitcoin предусматривает два механизма. Первый обеспечивает *поддержание сложности доказательства выполнения работы* (PoW). Если блоки добываются слишком быстро или слишком медленно, алгоритм Bitcoin, используемый всеми участниками, динамически адаптируется к условиям сети, увеличивая или уменьшая *сложность* PoW. При этом майнеры вынуждены находить дайджест блока, содержащий больше или меньше нулей.

ПРИМЕЧАНИЕ

Если сложность PoW такова, что дайджест блока должен начинаться с одного нуля, то вам придется перебрать 2^8 различных блоков (точнее, nonce, как было сказано ранее), чтобы найти действительный дайджест. Если увеличить это значение до двух нулей, то вам придется перебрать 2^{16} блоков. Время, которое потребуется для достижения этой цели, зависит от имеющихся у вас вычислительных мощностей и специализированного оборудования для более быстрого вычисления хешей. Сейчас алгоритм Bitcoin динамически изменяет сложность таким образом, чтобы новый блок добывался каждые 10 мин.

Второй механизм позволяет гарантировать то, что в случае возникновения развилки все участники будут двигаться в одинаковом направлении. Для этого существует правило, согласно которому они должны *продолжать цепочку с наибольшим количеством работы*. В техническом документе проекта Bitcoin, опубликованном в 2008 году, говорилось: «Самая длинная цепочка служит не только доказательством наблюдения последовательности событий, но и доказательством того, что она была произведена самым большим пулом процессорных мощностей», а это означает, что участники должны продолжать самую длинную цепочку. Позднее протокол был обновлен так, чтобы все участники продолжали цепочку с наибольшим суммарным объемом работы, но здесь это различие не имеет большого значения (рис. 12.6).

Я уже говорил о том, что алгоритм консенсуса системы Bitcoin не является BFT-протоколом, поскольку допускает появление форков. Таким образом, если вы ждете обработки своей транзакции, *не* стоит полагаться лишь на ее включение в блок! В случае появления форка такой блок может оказаться в проигрышной цепочке.

Для того чтобы удостовериться в выполнении вашей транзакции, требуются дополнительные гарантии. Большинство кошельков и обменных платформ ожидают, пока поверх вашего блока не появится несколько *подтверждающих блоков*.

Чем больше блоков будет добыто после того, который включает вашу транзакцию, тем меньше вероятность реорганизации соответствующей цепочки в связи с появлением форка.

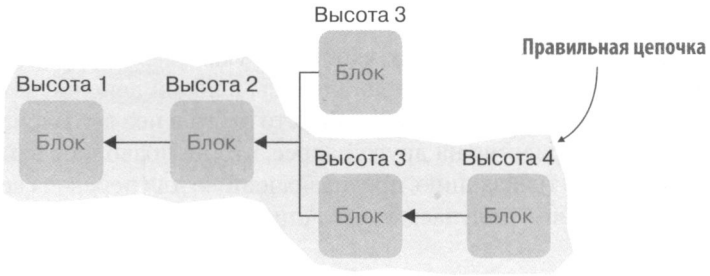


Рис. 12.6. Форк блокчейна: два майнера публикуют действительный блок на высоте 3 (то есть через три блока после генезис-блока). Позднее другой майнер добывает блок на высоте 4, который ссылается на второй блок на высоте 3. Поскольку в результате этого вторая цепочка оказывается длиннее, она считается действительной и именно ее майнерам следует продолжать. Обратите внимание на то, что стрелка, отходящая от блока, указывает на родительский блок

Как правило, для подтверждения транзакции бывает достаточно шести блоков, поэтому ее обработка занимает около часа. Тем не менее даже в этом случае система Bitcoin не дает стопроцентной гарантии того, что форк не возникнет после появления этих шести блоков. Если сложность майнинга хорошо отрегулирована, то все должно быть в порядке, а у нас есть основания полагать, что в системе Bitcoin дела обстоят именно так.

Сложность PoW в системе Bitcoin постепенно увеличивается по мере роста популярности этой криптовалюты. Сейчас она настолько высока, что большинство людей не могут позволить себе приобрести оборудование, необходимое для добычи блока. Сегодня большинство майнеров объединяются в так называемые *майнинг-пулы*, чтобы распределить работу по добыче блока и разделить между собой вознаграждение.

> После добычи блока 632 874 <...> ожидаемое количество выполненной работы в блокчейне Bitcoin превысило 292 двойных хеша SHA-256.
>
> *Питер Вуйле (http://mng.bz/aZNJ, 2020)*

Чтобы понять, в чем заключается разрушительность форков, рассмотрим следующий сценарий. Алиса покупает у вас бутылку вина, и вы ждете, пока она отправит вам 5 BTC со своего счета. Наконец видите новый блок на высоте 10 (то есть через десять блоков после генезис-блока), который включает ее транзакцию. Будучи осторожным человеком, вы решаете подождать, пока после него не появятся еще шесть блоков. Через некоторое время на высоте 16 возникает блок, который продолжает цепочку, содержащую ваш блок на высоте 10. Вы отправляете Алисе бутылку вина и завершаете сделку. Но на этом история не заканчивается.

Чуть позже буквально из ниоткуда на высоте 30 появляется блок, продолжающий другую цепочку, возникшую в результате разветвления, произошедшего всего за блок до вашего — на высоте 9. Поскольку новая цепочка оказывается длиннее, все участники принимают ее в качестве действительной. Предыдущая цепочка, в которой находились вы (начиная с блока на высоте 10), отбрасывается, и участники сети реорганизуют свои цепочки так, чтобы они указывали на новую, самую длинную цепочку. Как вы, вероятно, уже догадались, в новой цепочке нет ни одного блока, включающего транзакцию Алисы. Вместо этого в нее включена транзакция, переводящая все ее средства на другой адрес, что не позволяет вам повторно опубликовать исходную транзакцию, предназначенную для перевода ее средств на ваш адрес. Таким образом, получается, что Алиса фактически *дважды потратила* свои деньги.

В этом и состоит суть так называемой *атаки 51 %*. Ее название происходит от количества вычислительных мощностей, необходимых Алисе для выполнения атаки, — их ей потребовалось чуть больше, чем всем остальным. (На сайте https://crypto51.app есть интересная таблица, в которой указаны затраты на реализацию атаки 51 % на различные криптовалюты, основанные на механизме PoW.) Эти атаки — не что-то теоретическое! Они происходят в реальном мире. Например, в 2018 году с помощью атаки 51 % злоумышленнику удалось дважды потратить некоторое количество валюты Vertcoin.

> Злоумышленник, по сути, переписал часть истории в реестре, а затем, используя свое преобладающее количество мощностей, создал самую длинную цепочку, убедив остальных майнеров подтвердить эту новую версию блокчейна. Таким образом он смог совершить идеальное криптопреступление: дважды провести транзакции, оставив предыдущих получателей платежей с недействительными монетами.
>
> *Майкл Джей Кейси (Vertcoin's Struggle Is Real: Why the Latest Crypto 51 % Attack Matters, 2018)*

В 2019 году то же самое произошло с валютой Ethereum Classic (разновидностью Ethereum). Это привело к потерям более 1 млн долларов США и нескольким реорганизациям блокчейна более чем на 100 блоков в глубину. В 2020 году валюта Bitcoin Gold (разновидность Bitcoin) также подверглась атаке 51 %, которая привела к удалению из ее истории 29 блоков и двойной трате средств более чем на 70 000 долларов, произошедшей менее чем за два дня.

12.2.4. Уменьшение размера блока с помощью деревьев Меркла

Последний интересный аспект системы Bitcoin, о котором я хочу рассказать, связан с тем, как она сжимает часть доступной информации. Фактически ее блоки не содержат транзакций! Транзакции публикуются отдельно, а вместо них блок содержит один дайджест, который удостоверяет подлинность списка транзакций. Этим дайджестом мог бы быть просто хеш всех транзакций, содержащихся в блоке, но в данном случае все немного интереснее — дайджест представляет собой корень *дерева Меркла*.

Дерево Меркла — это дерево (структура данных), в котором каждый из внутренних узлов содержит хеш-значение конкатенации хешей двух его дочерних узлов. Чтобы лучше понять его суть, обратитесь к рис. 12.7.

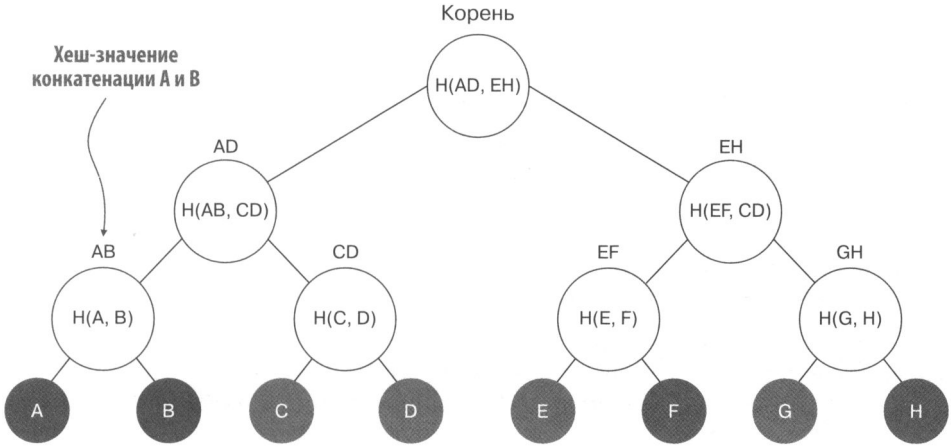


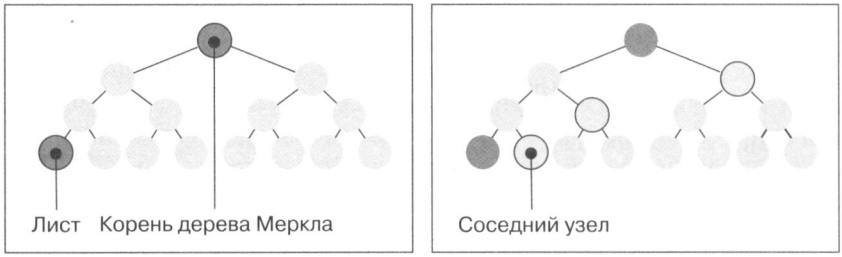
Рис. 12.7. Дерево Меркла — это структура данных, которая проверяет подлинность значений, содержащихся в ее листьях. В нем внутренний узел представляет собой хеш его дочерних элементов. Корневой хеш может использоваться для аутентификации всей структуры. H(...) на диаграмме представляет собой хеш-функцию, а входные данные, разделенные запятыми, могут конкатенироваться (при отсутствии двусмысленности)

Деревья Меркла — это весьма полезные структуры, которые используются во всевозможных реальных протоколах. Они позволяют сжать большой объем данных до небольшого значения фиксированного размера, содержащегося в корне дерева. При этом для восстановления хеша корня вам не обязательно задействовать все листья.

Представьте, что вам известен корень дерева Меркла благодаря его включению в блок Bitcoin и вы хотите знать, включена ли в этот блок транзакция (лист дерева). Если она присутствует в дереве, то я могу предоставить вам значения соседних узлов на пути к корню в качестве *доказательства принадлежности*, размер которого логарифмически зависит от глубины дерева. После этого вам останется лишь вычислить значения внутренних узлов вплоть до корня дерева, хешируя каждую их пару на этом пути. Объяснить это словами сложновато, поэтому я проиллюстрировал суть идеи на рис. 12.8.

Преимущество использования деревьев Меркла в блоке вместо непосредственного перечисления всех транзакций заключается в том, что они позволяют сократить объем информации, которую необходимо загрузить для выполнения простых запросов к блокчейну. Представьте, что вы хотите проверить, включена ли ваша недавняя транзакция в блок, не загружая при этом всю историю блокчейна Bitcoin. Для этого можно загрузить только заголовки блоков — они более легкие, так как не содержат транзакций, а затем попросить пира сообщить, в какой блок включена

ваша транзакция. Если такой блок есть, он должен быть в состоянии представить вам доказательство того, что транзакция находится в дереве, подтвержденное дайджестом, обнаруженным в заголовке блока.



1. Доказательство принадлежности листа при наличии знаний о корне

2. Используя соседние узлы, можно проверить, ведет ли путь к корню

Рис. 12.8. Зная корень дерева Меркла, можно проверить принадлежность листа этому дереву, восстановив хеш-значение корня из всех листьев. Для этого вам понадобятся все листья. На диаграмме это восемь дайджестов (при условии, что листья содержат хеши некоторых объектов). Есть более эффективный способ построения доказательства принадлежности, при котором вам потребуются только соседние узлы на пути от листа до корня, то есть всего четыре дайджеста, включая ваш лист. Проверяющий может задействовать эти соседние узлы для вычисления хешей всех недостающих узлов на пути к корню, пока не восстановит хеш корня и не посмотрит, соответствует ли его значение ожидаемому

О биткойне можно рассказать еще много интересного, но количество страниц в этой книге весьма ограничено. Поэтому в оставшейся части главы я расскажу о сфере криптовалют в целом и объясню принцип работы классических BFT-протоколов консенсуса.

12.3. ЭКСКУРСИЯ ПО МИРУ КРИПТОВАЛЮТ

Биткойн — первая успешная криптовалюта, которая до сих пор занимает самую значимую долю рынка и больше всех стоит, несмотря на существование сотен других криптовалют. Интересно то, что у биткойна было и есть множество проблем, которые были призваны решить другие криптовалюты (и некоторым из них это удалось). Еще интереснее то, что в криптовалютной сфере используются многие криптографические примитивы, которые до сих пор не имели широкого практического применения или даже не существовали! В следующих разделах мы рассмотрим проблемы, которые стали активно изучаться с момента появления биткойна.

12.3.1. Волатильность

В настоящее время большинство людей используют криптовалюты в качестве инструмента для спекуляций. Цена биткойна, безусловно, способствует этому, поскольку, как показала практика, она может легко измениться на несколько тысяч долларов в течение одного дня. Некоторые люди утверждают, что курс криптовалюты со

временем стабилизируется, но факт остается фактом: сейчас биткойн едва ли можно рассматривать в качестве обычной валюты. Разработчики других криптовалют экспериментируют с такой концепцией, как *стейблкойн*, привязывая цену своего токена к курсу существующей национальной валюты, например доллара США.

12.3.2. Время ожидания

Эффективность работы криптовалютной сети можно измерить разными способами. *Пропускная способность* — это количество транзакций, которые она способна обработать за 1 с. Например, пропускная способность сети Bitcoin довольно низкая и составляет всего семь транзакций в секунду. В то же время под *финализацией* понимается время, которое требуется для того, чтобы попавшая в блокчейн транзакция считалась завершенной. Из-за форков окончательная финализация в сети Bitcoin не достигается никогда. Считается, что примерно через час после включения транзакции в новый блок вероятность ее отмены становится приемлемой. Оба эти показателя сильно влияют на *время ожидания* — период, необходимый для выполнения транзакции с точки зрения пользователя. В сети Bitcoin этот период включает в себя несколько отрезков времени: затрачиваемое на создание транзакции, необходимое для ее распространения по сети, для ее включения в блок и, наконец, время ожидания подтверждения этого блока.

Решить эти связанные со скоростью проблемы позволяют BFT-протоколы, которые обычно обеспечивают финализацию за считаные секунды с гарантией отсутствия форков, а также пропускную способность порядка тысяч транзакций в секунду. Тем не менее иногда этого оказывается недостаточно, поэтому сейчас разрабатываются и другие технологии. Так называемые *протоколы второго уровня* призваны предложить дополнительные решения, которые могут обеспечить более быстрое проведение платежей вне блокчейна с периодическим сохранением прогресса в основном блокчейне, называемом сетью первого уровня.

12.3.3. Размер блокчейна

Другая распространенная проблема биткойна и других криптовалют заключается в быстром разрастании блокчейна до огромных размеров. Это отрицательно сказывается на удобстве использования, поскольку людям, желающим работать с криптовалютой (например, чтобы узнать баланс своего счета), приходится загружать всю цепочку для взаимодействия с сетью. Считается, что блокчейны криптовалют на основе BFT, обрабатывающие большое количество транзакций в секунду, легко накапливают терабайты данных в течение нескольких месяцев или даже недель. Для решения этой проблемы было предложено несколько решений.

Одно из самых интересных — протокол Mina, который не требует загрузки всей истории блокчейна для выяснения последнего состояния. Вместо этого он использует доказательства с нулевым разглашением (zero-knowledge proof, ZKP), о которых говорилось в главе 7 и которые будут подробно рассмотрены в главе 15, для сжатия всей истории в доказательство фиксированного размера в 11 Кбайт. Это особенно удобно для таких легких клиентов, как мобильные телефоны, которые обычно вынуждены доверять сторонним серверам для отправки запросов к блокчейну.

12.3.4. Конфиденциальность

Система Bitcoin обеспечивает *псевдоанонимность*, поскольку учетные записи привязаны только к открытым ключам. До тех пор пока никто не свяжет открытый ключ с личностью конкретного человека, его счет будет оставаться анонимным. Помните, что все переводы, выполненные с этого счета и на него, находятся в открытом доступе и на основе этой информации можно создавать социальные профили, чтобы понять, кто и с кем чаще всего взаимодействует и каким количеством валюты владеют эти люди.

Существует множество криптовалют, призванных решить эти проблемы с помощью ZKP или других методов. *Zcash* — одна из самых известных конфиденциальных криптовалют, поскольку ее транзакции позволяют шифровать адрес отправителя, адрес получателя и сумму сделки. И все это с помощью ZKP!

12.3.5. Энергоэффективность

Систему Bitcoin серьезно критикуют за то, что она потребляет слишком много электроэнергии. Действительно, по оценкам Кембриджского университета, количество энергии, затрачиваемой на добычу BTC, делает эту систему одним из 30 основных мировых энергопотребителей (если рассматривать ее в качестве страны). За год она потребляет больше энергии, чем такая страна, как Аргентина (февраль 2021-го, https://cbeci.org/). В то же время BFT-протоколы не полагаются на механизм PoW, что позволяет им избавиться от таких огромных накладных расходов. Именно поэтому современные криптовалюты избегают использования механизма консенсуса, основанного на PoW, и даже такие важные криптовалюты, как Ethereum, объявляют о планах перехода на применение более экологичных протоколов консенсуса. Прежде чем завершить данную главу, давайте познакомимся с криптовалютами, основанными на протоколах консенсуса BFT.

12.4. DIEMBFT — ВИЗАНТИЙСКИЙ ОТКАЗОУСТОЙЧИВЫЙ ПРОТОКОЛ КОНСЕНСУСА

Многие современные криптовалюты отказались от применения механизма PoW в пользу более экологичных и эффективных протоколов консенсуса. Большинство таких протоколов основано на классических протоколах консенсуса BFT, которые чаще всего представляют собой различные варианты оригинального протокола PBFT. В этом, последнем, разделе я проиллюстрирую принцип работы таких BFT-протоколов на примере криптовалюты Diem.

Diem (изначально называвшаяся Libra) — это цифровая валюта, анонсированная Facebook в 2019 году и управляемая Diem Association — организацией, объединяющей университеты и некоммерческие компании, стремящиеся к созданию открытой и глобальной платежной сети. Особенностью Diem является то, что она обеспечена реальными деньгами в виде резерва национальных валют. Это позволяет данной цифровой валюте поддерживать стабильный курс, в отличие от ее предшественника — биткойна. Для обеспечения безопасной и открытой работы этой

платежной сети используется протокол консенсуса BFT под названием *DiemBFT*, который представляет собой вариант протокола HotStuff. В этом разделе рассмотрим принцип его работы.

12.4.1. Безопасность и живость — два свойства протокола консенсуса BFT

Протокол консенсуса BFT призван обеспечить два свойства даже при существовании определенного процента злоумышленников среди его участников. Этими свойствами являются:

- *безопасность* (safety) — невозможность согласования противоречивых состояний. Это означает невозможность (или пренебрежимо малую вероятность) возникновения форков;
- *живость* (liveness) — все отправленные транзакции в конечном итоге обрабатываются. Другими словами, никто не может помешать протоколу делать свое дело.

Обратите внимание на то, что участник считается злоумышленником (также называемым *византийцем*), если он не ведет себя в соответствии с протоколом. Это может означать, что он ничего не делает, выполняет шаги протокола в неверном порядке, не соблюдает обязательные правила, призванные обеспечить отсутствие форков, и т. д.

Обычно BFT-протоколы консенсуса довольно легко обеспечивают безопасность, в то время как с живостью все несколько сложнее. Действительно, хорошо известна теорема Фишера — Линча — Патерсона, описанная в работе *Impossibility of distributed consensus with one faulty process* 1985 года, согласно которой ни один *детерминированный* протокол консенсуса не способен выдержать отказы в *асинхронной* сети, в которой доставка сообщений может занимать любое количество времени. Большинство BFT-протоколов справляется с этой проблемой, считая сеть в некоторой степени *синхронной* (и действительно, ни один протокол не может быть полезным, если ваша сеть не работает в течение длительного периода времени) или привнося в алгоритм элемент случайности.

По этой причине протокол DiemBFT никогда не допускает возникновения форков, даже когда сеть работает в экстремальных условиях. Кроме того, он всегда движется вперед, даже когда некоторые части сети не могут связаться с другими ее частями, при условии, что в итоге сеть восстанавливается и работает стабильно в течение довольно длительного периода времени.

12.4.2. Раунд в протоколе DiemBFT

Diem работает в эксклюзивной среде, где участники, называемые *валидаторами*, известны заранее. Протокол выполняет раунды (циклы) в строго возрастающем порядке (раунд 1, 2, 3 и т. д.), в ходе которых валидаторы по очереди предлагают блоки транзакций. В рамках каждого раунда происходит следующее.

1. Валидатор, выбранный в качестве лидера (детерминистски), собирает некоторое количество транзакций, группирует их в новый блок, продолжающий цепочку, затем подписывает этот блок и отправляет его остальным валидаторам.

2. Получив предложенный блок, другие валидаторы могут проголосовать за его сертификацию, подписав его и отправив подпись лидеру следующего раунда.
3. Если лидер следующего раунда получает достаточно голосов в поддержку этого блока, он может объединить их в так называемый *сертификат кворума* (СК), который сертифицирует данный блок, и использовать СК для предложения нового блока (в следующем раунде), продолжающего соответствующую цепочку.

Если в системе Bitcoin блок содержит только хеш родительского блока, то в DiemBFT — также ряд подписей под этим хешем. (Количество подписей очень важно, но об этом мы поговорим чуть позже.)

Учтите, если валидаторы не видят предложенного блока во время раунда (например, из-за того что лидер отошел от компьютера), они могут объявить тайм-аут и предупредить других валидаторов о том, что ничего не произошло. В этом случае запускается следующий раунд, и новый лидер может предложить блок, следующий за последним сертифицированным блоком (рис. 12.9).

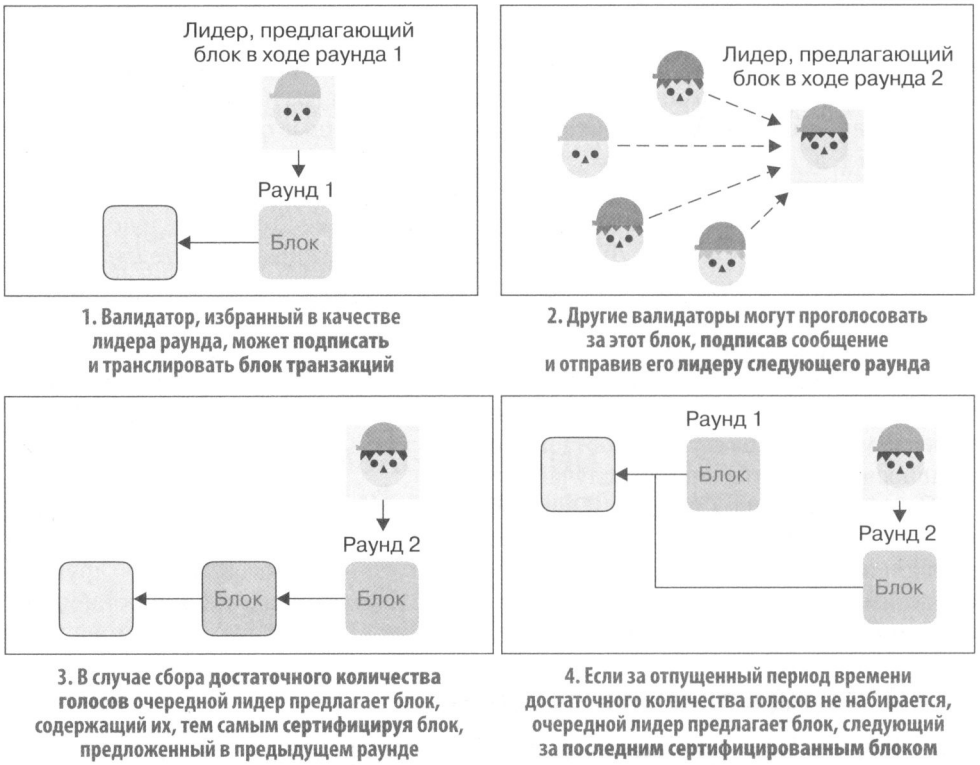


Рис. 12.9. Каждый раунд протокола DiemBFT начинается с того, что выбранный лидер предлагает блок, следующий за последним из блоков, которые видели участники. Другие валидаторы могут проголосовать за этот блок, отправив свой голос лидеру следующего раунда. Если лидер следующего раунда набирает достаточно голосов для формирования сертификата кворума, он может предложить новый блок, содержащий СК и следующий за ранее виденным блоком

12.4.3. Какое количество нечестных участников допускает протокол

Если мы хотим, чтобы наш протокол выдерживал присутствие f злонамеренных валидаторов (даже вступивших в сговор), то в протоколе DiemBFT должны участвовать не менее $3f + 1$ валидаторов (другими словами, на f злоумышленников должно приходиться не менее $2f + 1$ честных валидаторов). Пока это условие выполняется, протокол обладает свойствами безопасности и живости.

Учитывая это, СК могут быть сформированы только большинством голосов честных валидаторов, а это $2f + 1$ подписей при общем количестве участников $3f + 1$. Чтобы лучше понять, как эти числа влияют на уровень доверия к результатам голосования, обратитесь к рис. 12.10.

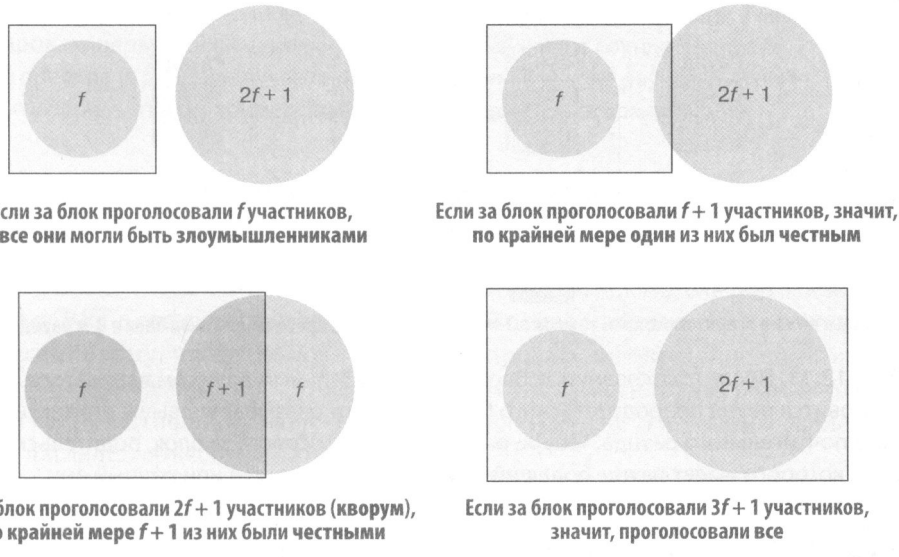


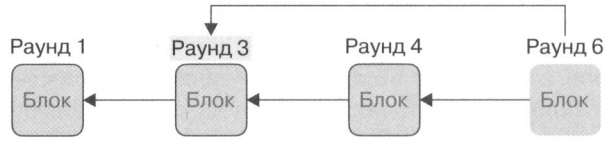
Рис. 12.10. Для того чтобы протокол DiemBFT был безопасным (не допускал форков) и живым (двигался вперед), по крайней мере две трети валидаторов должны быть честными. Другими словами, этот протокол может вынести присутствие f нечестных валидаторов при условии, что как минимум 2f + 1 валидаторов будут честными. Сертифицированный блок получает не менее 2f + 1 голосов, так как это наименьшее число голосов, которое может представлять большинство честных валидаторов

12.4.4. Правила голосования в DiemBFT

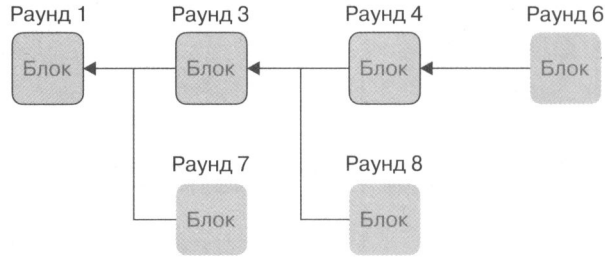
Валидаторы должны постоянно следовать двум правилам голосования, чтобы не считаться византийцами.

1. Они не могут голосовать в прошлом (например, если вы только что закончили голосование в раунде 3, то можете голосовать только в раунде 4 и выше).
2. Они могут голосовать только за тот блок, который следует за блоком, предложенным в ходе их предпочтительного раунда или выше.

Что такое *предпочтительный раунд* (preferred round)? По умолчанию это нулевой раунд, но если вы проголосуете за блок, который следует за блоком, который следует за блоком, то есть за блок, имеющий прародителя, то раунд этого прародительского блока становится вашим предпочтительным раундом при условии, что предыдущий предпочтительный раунд не является более поздним. Чтобы лучше в этом разобраться, обратитесь к рис. 12.11.



Если я проголосую за блок, предложенный в 6-м раунде, моим предпочтительным раундом станет 3-й



Тогда я не смогу проголосовать за блок в 7-м раунде, но смогу проголосовать за блок в 8-м раунде

Рис. 12.11. После голосования за блок предпочтительным раундом валидатора становится раунд прародительского блока, если тот оказывается выше его текущего предпочтительного раунда. Теперь он может проголосовать за блок, родительский блок которого имеет раунд, больший чем предпочтительный или равный ему

12.4.5. Когда транзакции считаются завершенными

Обратите внимание на то, что сертифицированные блоки еще не являются финализированными или *зафиксированными*. Никому не следует полагать, что транзакции, содержащиеся в блоках, находящихся в состоянии ожидания, не могут быть отменены. Блоки и содержащиеся в них транзакции можно считать финализированными только после выполнения *правила фиксации*. Согласно данному правилу (рис. 12.12) блок и все ожидающие блоки, за которыми он следует, становятся зафиксированными, если:

- этот блок начинает цепочку из трех блоков, предложенных в ходе *последовательных раундов* (например, в раундах 1, 2 и 3);
- последний блок цепочки из трех блоков становится сертифицированным.

Итак, мы в общих чертах рассмотрели принцип работы протокола. Но дьявол, разумеется, кроется в деталях.

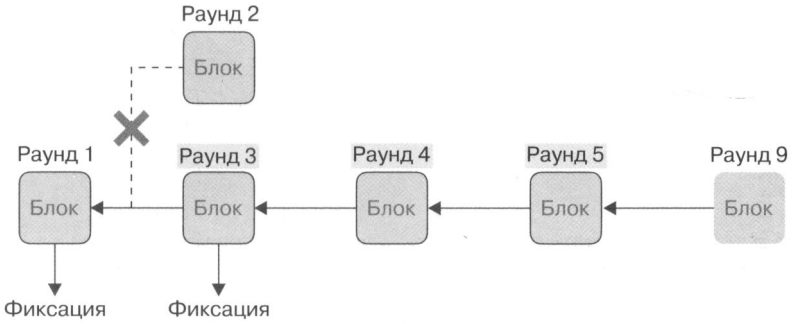


Рис. 12.12. Три последовательных раунда (3, 4, 5-й) формируют цепочку сертифицированных блоков. Любой валидатор, наблюдающий сертификацию последнего блока, предложенного в ходе раунда 5, с помощью СК раунда 9 может зафиксировать первый блок цепочки, предложенный в ходе раунда 3, а также всех его предков (в данном случае блок, предложенный в ходе раунда 1). Любые конфликтующие ветви (например, блок раунда 2) при этом отбрасываются

12.4.6. Интуитивное понимание безопасности протокола DiemBFT

Хотя я рекомендую вам ознакомиться с одностраничным доказательством безопасности DiemBFT в соответствующем техническом документе, хочу также посвятить пару страниц тому, чтобы помочь вам сформировать интуитивное понимание того, как она обеспечивается. Во-первых, как вы уже заметили, два разных блока не могут быть сертифицированы в течение одного раунда. Это важное свойство, которое наглядно объяснено на рис. 12.13.

Учитывая, что в ходе того или иного раунда может быть сертифицирован только один блок, можно упростить то, как мы говорим о блоках, сказав, что блок 3 соответствует раунду 3, блок 6 — раунду 6 и т. д. Теперь посмотрите на рис. 12.14 и попробуйте понять, почему один, два или три сертифицированных блока, соответствующих непоследовательным раундам, нельзя зафиксировать, не рискуя возникновением форка.

Удалось ли вам найти ответы для всех сценариев? Если вкратце, то все сценарии, за исключением последнего, предусматривают возможность появления блока, следующего за блоком, предложенным в ходе раунда 1. Этот более поздний блок создает новую ветвь, которая может быть продлена дальше в соответствии с правилами протокола консенсуса. Если это произойдет, то блок 5 и следующие за ним будут отброшены, а более ранняя ветвь — зафиксирована. В сценариях 1 и 2 это может произойти из-за того, что предлагающий не увидел предыдущих блоков. В сценарии 3 более ранний блок может появиться позднее, чем ожидалось, например, из-за задержек в сети или, что еще хуже, из-за того, что валидатор придерживает его до нужного момента (рис. 12.15).

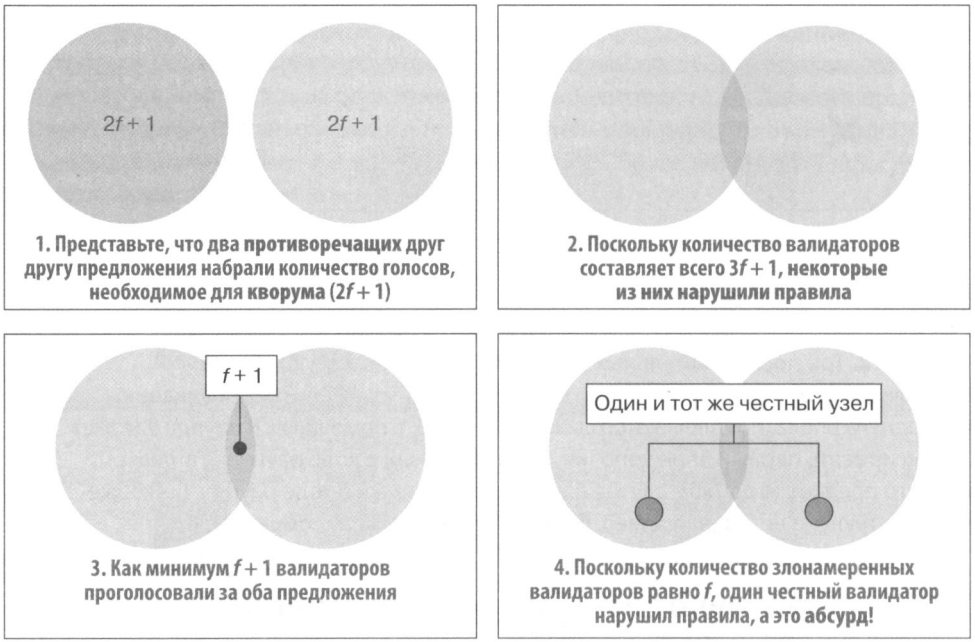


Рис. 12.13. Если предположить, что из 3f + 1 валидаторов протокола злонамеренными могут быть не более f валидаторов, а сертификат кворума требует сбора 2f + 1 подписанных голосов, то в каждом раунде может быть только один сертифицированный блок. На диаграмме приведено доказательство от противного, показывающее, что некое событие не может произойти, поскольку оно противоречит исходным допущениям

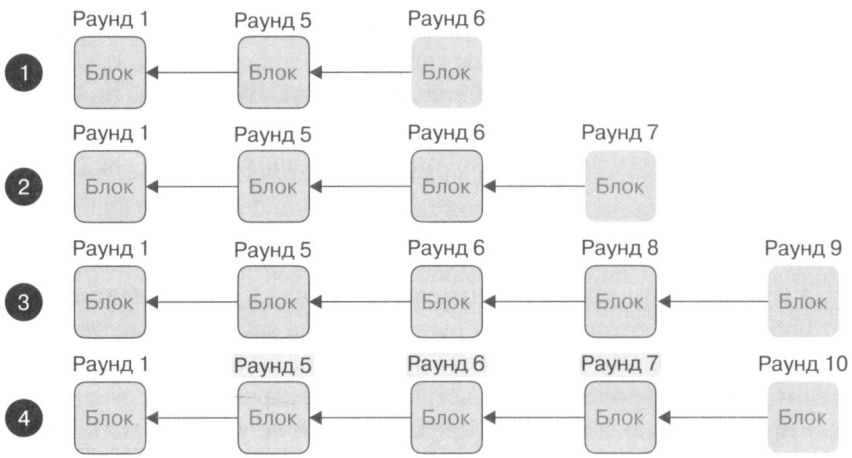


Рис. 12.14. В первых трех сценариях фиксация блока 5 может привести к возникновению форка. И только в сценарии 4 фиксация блока 5 безопасная. Можете ли вы сказать почему?

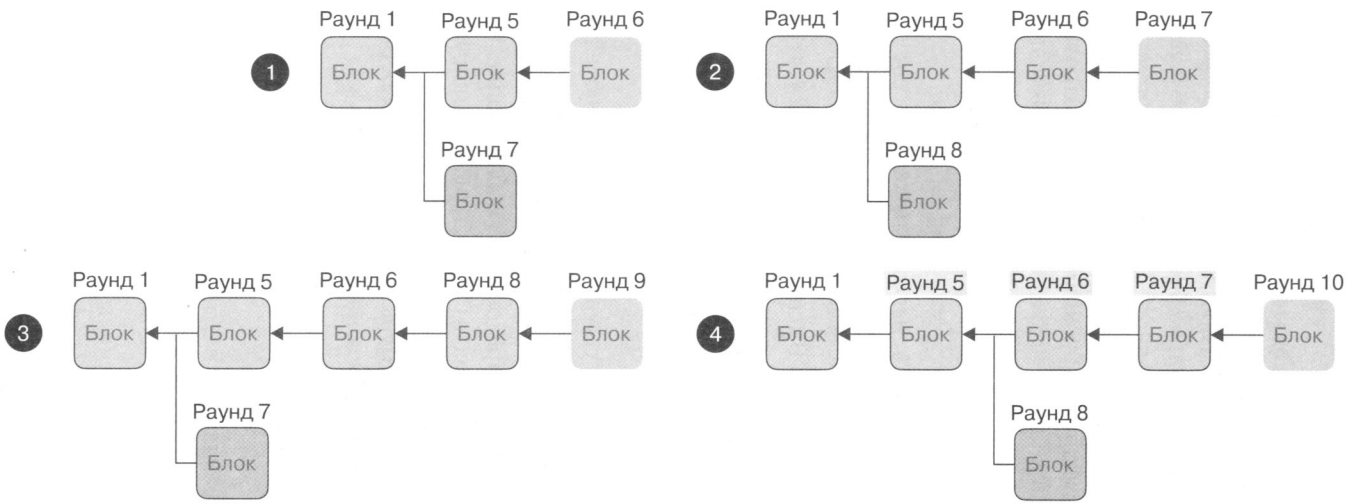


Рис. 12.15. Дополнение к схеме, приведенной на рис. 12.14. Все сценарии, кроме последнего, допускают возникновение параллельной цепочки, которая в итоге может быть зафиксирована, а ветвь, начинающаяся с блока 5, — отброшена. В последнем сценарии мы видим цепочку из трех сертифицированных блоков, предложенных в трех последовательных раундах. Это означает, что за блок 7 проголосовало большинство честных валидаторов, которые, в свою очередь, обновили свой предпочтительный раунд до раунда 5. После этого ни один блок не сможет сформировать ответвление перед блоком 5, получив при этом сертификат кворума. Худшее, что может произойти, — какой-нибудь блок будет следовать за блоком 5 или 6, что приведет к одному и тому же результату — фиксации блока 5

РЕЗЮМЕ

- Криптовалюты предназначены для децентрализации платежной сети с целью избавления от единой точки отказа.
- Для достижения всеми участниками согласия относительно состояния криптовалюты мы можем использовать алгоритмы консенсуса.
- Византийские отказоустойчивые (BFT) протоколы консенсуса были изобретены в 1982 году и с тех пор эволюционировали, становясь все более быстрыми и простыми для понимания.
- Для обеспечения работы протоколов консенсуса BFT требуется известный фиксированный список участников (эксклюзивная сеть). Протоколы могут составлять такой список, используя механизм PoA (доказательство полномочий), или отбирать участников динамически с помощью механизма PoS (доказательство доли владения), то есть исходя из количества принадлежащей им валюты.
- Алгоритм консенсуса системы Bitcoin, известный как консенсус Накамото, использует доказательство выполнения работы (PoW) для подтверждения правильности цепочки и допускает участие всех желающих (общедоступная сеть).
- Суть работы механизма PoW системы Bitcoin сводится к тому, что ее участники, называемые майнерами, вычисляют множество хешей, пытаясь отыскать значения с определенными префиксами. Успешное нахождение правильного дайджеста позволяет майнеру принять решение относительно следующего блока транзакций и получить вознаграждение, а также транзакционные комиссии.
- Счета в системе Bitcoin представляют собой просто пары ключей ECDSA, которые генерируются с помощью кривой secp256k1. Пользователь определяет баланс своего счета на основе неизрасходованных транзакционных выходов (UTXO). Таким образом, транзакция представляет собой подписанное сообщение, санкционирующее перемещение ряда старых транзакционных выходов на адреса, связанные с другими открытыми ключами.
- Система Bitcoin использует деревья Меркла для уменьшения размера блока и упрощения процесса проверки включения транзакций в блок.
- Стейблкойнами называются криптовалюты, курс которых стабилизируется путем привязки стоимости токена к стоимости национальной валюты, например доллара США.
- Криптовалютные сети применяют так называемые протоколы второго уровня, чтобы уменьшить время ожидания за счет обработки транзакций вне блокчейна с периодическим сохранением прогресса в нем.
- Доказательства с нулевым разглашением (ZKP) используются во многих блокчейн-приложениях, например в Zcash — для обеспечения конфиденциальности и в Coda — для сжатия всего блокчейна до короткого доказательства действительности.

- Diem — это стейблкойн, использующий BFT-протокол консенсуса под названием DiemBFT. Он остается безопасным (не допускающим появления форков) и живым (движущимся вперед) до тех пор, пока среди $3f + 1$ участников число злоумышленников не превышает f.
- Протокол DiemBFT предусматривает проведение раундов, в ходе которых участник предлагает блок транзакций, следующий за предыдущим блоком. Затем другие участники могут проголосовать за этот блок. В случае сбора достаточного количества голосов ($2f + 1$) создается сертификат кворума.
- В рамках протокола DiemBFT блоки и содержащиеся в них транзакции финализируются при выполнении условий правила фиксации (формирования цепочки из трех сертифицированных блоков, предложенных в трех последовательных раундах). Когда это происходит, выполняется фиксация первого блока цепочки и тех блоков, которые находятся перед ним.

13 Аппаратная криптография

В этой главе

- ✓ Проблемы криптографии во враждебных средах.
- ✓ Аппаратные решения для увеличения издержек злоумышленника.
- ✓ Атаки по сторонним каналам и программные средства защиты.

Криптографические примитивы и протоколы часто описываются как изолированные строительные блоки, работающие в какой-то далекой и свободной от противников галактике. Это предположение нереалистично и часто опровергается практикой. В реальном мире криптографические инструменты применяются в различных средах и подвергаются всевозможным угрозам. В этой главе рассмотрим наиболее экстремальные сценарии — *среды с высокой степенью враждебности*, а также способы защиты своих ключей и данных, которые можно использовать в этих ситуациях. (Спойлер: для этого вам понадобится специализированное оборудование.)

13.1. МОДЕЛЬ ЗЛОУМЫШЛЕННИКА В СОВРЕМЕННОЙ КРИПТОГРАФИИ

В основе современной компьютерной и сетевой безопасности лежит предположение о существовании домена, которому мы можем доверять. Например, если мы шифруем данные для их передачи через Интернет, то обычно считаем, что компьютер, на котором происходит шифрование, не скомпрометирован и существует некая другая конечная точка, в которой данные могут быть безопасно расшифрованы.

Джоанна Рутковска (Intel x86 considered harmful, 2015)

Раньше суть криптографии можно было выразить фразой: «Алиса хочет зашифровать сообщение, адресованное Бобу, так, чтобы Ева не смогла его перехватить». Сегодня она формулируется скорее так: «Алиса хочет зашифровать сообщение, адресованное Бобу, но ее устройство скомпрометировано». Это совершенно иная модель злоумышленника, которая обычно не предусматривается в теоретической криптографии. Чтобы пояснить, что именно я имею в виду, приведу несколько примеров.

- Использование кредитной карты в банкомате, к которому может быть прикреплен *скиммер* — считывающее устройство, с помощью которого злоумышленник копирует данные банковской карты (рис. 13.1).
- Загрузка на мобильный телефон приложения, компрометирующего его операционную систему (ОС).
- Размещение веб-сервера на сервисе виртуального хостинга, при котором злоумышленник может использовать ту же машину, что и вы.
- Управление особо секретной информацией в центре обработки данных, к которому имеют доступ иностранные шпионы.

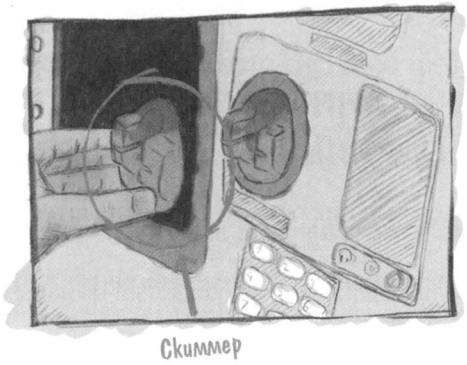


Рис. 13.1. Скиммер — это вредоносное устройство, которое может прикрепляться к считывателям карт банкоматов или платежных терминалов для копирования данных, содержащихся на магнитной полосе банковских карт. Обычно это номер счета, срок действия карты и другие метаданные, необходимые при оплате покупок в Интернете или платежных терминалах. Иногда скиммеры дополняются скрытой камерой, что позволяет мошеннику использовать банкоматы и платежные терминалы для кражи PIN-кодов

Все эти примеры демонстрируют современное применение криптографии в такой модели угроз, которую многие криптографы игнорируют или совершенно не осознают. Действительно, большинство криптографических примитивов, о которых вы читаете в литературе, предполагают, что, например, Алиса полностью контролирует свою среду выполнения, а злоумышленники получают возможность реализовать атаку посредника лишь тогда, когда шифротекст (подпись, открытый ключ или что-то еще) покидает ее компьютер и попадает в сеть. Однако

в современной реальности криптография применяется в моделях, отличающихся гораздо более высокой степенью враждебности.

ВНИМАНИЕ

Безопасность представляет собой продукт ваших предположений и ожиданий от потенциального злоумышленника. И если предположения неверны, вас не ждет ничего хорошего.

Для того чтобы примирить теоретическую криптографию с фактом существования этих более сильных злоумышленников, реальные приложения вынуждены идти на *компромисс*. Другими словами, они пытаются усложнить злодеям жизнь. Безопасность таких систем часто оценивается в терминах *издержек* (то есть количества ресурсов, которые вынужден потратить злоумышленник для взлома), а не вычислительной сложности.

Многое из того, что вы узнаете в этой главе, будет связано с *несовершенной* криптографией, которая в реальном мире называется *глубоко эшелонированной защитой*. Вам предстоит познакомиться со множеством новых решений и аббревиатур, придуманных различными производителями при участии их маркетологов и продавцов. Начнем с такого понятия, как доверенные системы в недоверенных средах.

13.2. НЕДОВЕРЕННЫЕ СРЕДЫ — АППАРАТНОЕ ОБЕСПЕЧЕНИЕ СПЕШИТ НА ПОМОЩЬ

Атаковать систему можно различными способами. Как правило, атаки делятся на две категории:

- *атаки на программное обеспечение*, использующие код, запущенный на вашем устройстве;
- *атаки на аппаратное обеспечение*, предполагающие физическое присутствие злоумышленника рядом с вашим устройством.

Хотя в предыдущих главах я уже неоднократно рассказывал об атаках на программное обеспечение и способах борьбы с ними, от некоторых нападений такого рода гораздо легче защититься с помощью аппаратных решений. Например, если генерировать и использовать криптографические ключи на отдельном устройстве, подключенном к компьютеру, то поразивший компьютер вирус не сможет их извлечь.

С атаками на аппаратное обеспечение все гораздо сложнее, поскольку, получив доступ к устройству, злоумышленник может делать практически все, что захочет: произвольно изменять хранящиеся на диске данные, направлять лазеры на определенные места для получения ошибочного результата вычислений (то есть реализовать так называемую *атаку методом внесения неисправностей*), вскрывать микросхемы, чтобы добраться до их частей, использовать микроскопы с фокусированным ионным пучком для обратного проектирования компонентов и т. д. Для

таких мотивированных злоумышленников пределом является лишь небо, и защититься от них очень сложно. Как правило, существующие решения предполагают добавление максимально возможного количества слоев защиты для усложнения задачи злодея. То есть все сводится к повышению его издержек!

АТАКА ЗЛОЙ ГОРНИЧНОЙ

Не все злоумышленники, атакующие аппаратное обеспечение, одинаковы. Некоторые из них получают возможность провести длительное время наедине с вашими устройствами, тогда как у других оно может быть очень ограниченным. Представьте следующий сценарий: вы оставляете свой телефон или ноутбук без присмотра в гостиничном номере, и тут приходит злонамеренная горничная, включает устройство, использует малобюджетный готовый инструмент для модификации системы и возвращает устройство на место. В литературе это называется *атакой злой горничной* и может быть распространено на многие ситуации, например провоз устройств в регистрируемом багаже, хранение секретных ключей в незащищенном центре обработки данных и т. д.

Разумеется, не всем системам требуется защита от мощнейших аппаратных атак и не все приложения сталкиваются с одинаковым уровнем угроз. Для разных контекстов существуют различные аппаратные решения, и именно пониманию различий между ними будет посвящена остальная часть данного раздела.

13.2.1. Криптография белого ящика — это плохая идея

Прежде чем переходить к применению аппаратных решений для недоверенных сред, почему бы не использовать программные решения? Может ли криптография предоставить примитивы, не допускающие утечки собственных ключей?

Именно этим занимается так называемая *криптография белого ящика* — область криптографии, которая пытается запутать криптографическую реализацию с помощью используемого в ней ключа. Цель состоит в том, чтобы предотвратить извлечение ключа сторонними наблюдателями. Злоумышленник получает исходный код некоторой реализации алгоритма AES с фиксированным ключом, который прекрасно шифрует и расшифровывает данные, но при этом настолько хорошо смешан с реализацией, что извлечь его из алгоритма слишком сложно. По крайней мере, в теории. На практике ни один из опубликованных криптоалгоритмов белого ящика не был признан безопасным, и большинство коммерческих решений имеют закрытый исходный код именно по этой причине.

ПРИМЕЧАНИЕ

Безопасность через неясность и обфускация (приведение кода в нечитаемый вид) — это методы, которые обычно не одобряются из-за отсутствия доказательств их эффективности. Тем не менее в реальном мире они занимают определенное место и иногда могут использоваться для сдерживания противника.

В целом криптография белого ящика — это большая индустрия, которая продает сомнительные продукты предприятиям, нуждающимся в решениях для *управления цифровыми правами* (digital rights management, DRM) — инструментах, контролирующих уровень доступа покупателя к приобретенному продукту. Например, такие решения могут использоваться в аппаратном обеспечении для воспроизведения фильмов, купленных в магазине, или в программном обеспечении для воспроизведения фильмов, которые вы смотрите через потоковый сервис. На самом деле DRM-решения не столько защищают от атак, сколько усложняют жизнь своим пользователям. Если серьезно, то существует целое направление криптографии под названием *«обфускация неразличимости»* (indistinguishability obfuscation, iO), направленное на криптографическое решение этой же задачи. iO — это сугубо теоретическая, непрактичная и не очень хорошо развитая область исследований. Посмотрим, что из этого выйдет, но я бы не стал возлагать на нее больших надежд.

13.2.2. Они в вашем бумажнике: смарт-карты и защищенные элементы

Криптография белого ящика имеет недостатки, но на практике оказывается самым лучшим программным решением для защиты от сильных противников. Теперь давайте рассмотрим аппаратные решения. (Спойлер: сейчас все станет гораздо сложнее и запутаннее.) Если область программной криптографии показалась вам беспорядочной и переполненной различными стандартами и способами решения одной и той же задачи, подождите, пока не узнаете, что происходит в области аппаратной криптографии, где выдуманные термины используются по-разному, а количество стандартов уже разрослось так же сильно, как в области программной криптографии, если не сильнее.

Чтобы понять, что представляют собой все эти аппаратные решения и чем они отличаются друг от друга, начнем с истории. *Смарт-карты* (или «умные» карты) — небольшие чипы, обычно упакованные в пластиковые карты (например, банковские), — были изобретены в начале 1970-х годов благодаря достижениям в области микроэлектроники. Создание смарт-карт позволило снабдить каждого человека карманным компьютером. Действительно, современная смарт-карта оснащена собственным процессором, различными типами программируемой и непрограммируемой памяти (ROM, RAM и EEPROM), входами и выходами, аппаратным генератором случайных чисел (также называемым TRNG, как сказано в главе 8) и т. д.

«Умными» эти карты называются потому, что на них можно запускать программы, чего нельзя сказать об обычных пластиковых картах, способных лишь хранить на магнитной полосе данные, которые можно легко скопировать с помощью упомянутых ранее скиммеров. Большинство смарт-карт способны выполнять небольшие приложения. Самым популярным стандартом, поддерживаемым смарт-картами, является *JavaCard*, позволяющий разработчикам писать Java-подобные приложения.

Чтобы воспользоваться смарт-картой, ее нужно сначала активировать, вставив в считывающее устройство. Недавно эти карты были дополнены технологией NFC (Near Field Communication — связь на ближнем расстоянии) для достижения того же результата с помощью радиочастот. Эта технология позволяет применять карту, поднося ее к считывающему устройству, но не касаясь его.

БАНКИ И УСТАРЕВШИЕ КРИПТОГРАФИЧЕСКИЕ МЕТОДЫ

Кстати, банки используют смарт-карты для хранения уникального секрета, который может подтвердить то, что данная карта действительно является той самой, выданной конкретному клиенту. Вы можете подумать, что это реализуется с помощью криптографии с открытым ключом, однако банковская отрасль застряла в прошлом и по-прежнему использует симметричную криптографию. Так происходит из-за огромного количества устаревшего программного и аппаратного обеспечения, которое до сих пор не выведено из эксплуатации.

Если говорить точнее, то в большинстве банковских карт хранится симметричный ключ алгоритма *triple-DES* (3DES), который представляет собой старый 64-битный блочный шифр, призванный сделать безопасным небезопасный стандарт шифрования данных (Data Encryption Standard, DES). Этот алгоритм используется не для шифрования, а для получения MAC (кода аутентификации сообщения) по некоторому запросу. Банк, владеющий текущим симметричным ключом 3DES каждого клиента, может проверить полученное значение MAC. Это отличный пример, демонстрирующий то, что в реальном мире устаревшие криптографические алгоритмы используются повсеместно и с определенной долей риска. (Именно поэтому ротация ключей — такая важная концепция, и именно поэтому вам следует периодически менять свои банковские карты.)

Устройство смарт-карт сочетает в себе ряд физических и логических приемов для предотвращения наблюдения за средой выполнения и частями памяти (хранящими секретные данные), их извлечения и модификации. Существует множество атак, направленных на взлом этих карт и аппаратных устройств в целом. Эти атаки можно разделить на три категории.

- *Неагрессивные, или неинвазивные, атаки* — это атаки, не затрагивающие целевое устройство. Например, дифференциальная атака по энергопотреблению (differential power analysis, DPA) предполагает оценку энергопотребления смарт-карты при одновременном выполнении криптографических операций для извлечения ключей.
- *Полуагрессивные, или полуинвазивные, атаки* — это атаки, в ходе которых для установки эксплойтов осуществляется доступ к поверхности чипа без его повреждения. Например, при реализации дифференциальной атаки по ошибкам вычислений (differential fault analysis, DFA) используются тепло, лазеры

и другие методы модификации процесса выполнения запущенной на смарт-карте программы с целью кражи ключей.

- *Агрессивные, или инвазивные, атаки* — это атаки, в ходе которых микросхема вскрывается для исследования или модификации самого кремниевого кристалла с целью изменения или раскрытия секретов работы чипа. Последствия подобных атак более заметны, поскольку они могут привести к повреждению устройства и даже полностью вывести его из строя.

Тот факт, что микросхемы очень малы и плотно упакованы, может затруднить реализацию атак. Однако для защиты специализированных аппаратных средств обычно применяются дополнительные слои материалов, предотвращающие их распаковку и физическое исследование, а также аппаратные методы повышения вероятности того, что при реализации известных атак будут допущены ошибки.

Смарт-карты очень быстро завоевали популярность, и всем стало очевидно, что наличие такого защищенного черного ящика в других устройствах тоже может оказаться полезным. Так возникла концепция *защищенного элемента* (secure element) — устойчивого к взлому микроконтроллера, который может вставляться в устройство (как, например, SIM-карта в вашем телефоне, необходимая для доступа к сети оператора связи) или встраиваться непосредственно в микросхемы и материнские платы (как защищенный элемент, прикрепленный к NFC-чипу в iPhone, предназначенному для платежей). Иными словами, защищенный элемент — это небольшая отдельная часть аппаратного обеспечения, предназначенная для защиты ваших секретов и обеспечения их безопасного использования в криптографических операциях.

Концепция защищенного элемента играет важную роль в обеспечении безопасности криптографических операций в сфере *Интернета вещей* (Internet of Things, IoT), которая охватывает устройства, способные взаимодействовать друг с другом, например кредитные карты, телефоны, биометрические паспорта, ключи от гаража, датчики для «умного» дома и т. д. Все решения, описанные в этом разделе, можно рассматривать как защищенные элементы, реализованные в различных формфакторах, использующие разные технологии для достижения практически одного и того же результата, но обеспечивающие разную скорость работы и уровень безопасности.

Основные определения и стандарты, касающиеся защищенных элементов, были разработаны некоммерческой ассоциацией Global Platform, созданной для облегчения взаимодействия между различными поставщиками и системами. Существуют и другие стандарты и сертификаты, фокусирующиеся на требованиях к безопасности этих элементов, которые разрабатываются такими органами стандартизации, как Common Criteria (CC), NIST или EMV (Europay, Mastercard и Visa).

Поскольку защищенные элементы являются секретными компонентами, их интеграция в ваш продукт означает, что вам придется подписывать соглашения о неразглашении и использовать аппаратное и микропрограммное обеспечение с закрытым исходным кодом. В контексте многих проектов это может восприниматься как серьезное ограничение прозрачности, но оно вполне понятно, учитывая то, что безопасность этих микросхем отчасти обеспечивается тем, что их внутреннее устройство неизвестно.

13.2.3. Излюбленные банками аппаратные модули безопасности

Аппаратный модуль безопасности (hardware security module, HSM) представляет собой более крупный и быстрый защищенный элемент, и, подобно некоторым защищенным элементам, часть HSM способна выполнять произвольный код. Однако так бывает не всегда. Некоторые HSM отличаются небольшими размерами (как, например, YubiHSM — крошечный USB-ключ, напоминающий YubiKey), и термин «аппаратный модуль безопасности» разные люди могут использовать для обозначения разных вещей.

Многие могут возразить, что все аппаратные решения, рассмотренные до сих пор, представляют собой ту или иную форму HSM-модуля, а защищенные элементы — это просто HSM, специфицированные ассоциацией GlobalPlatform, тогда как TPM (Trusted Platform Module — доверенный платформенный модуль) — это HSM, специфицированный организацией Trusted Computing Group. Но чаще всего, когда люди говорят о HSM, они имеют в виду нечто большое.

HSM обычно классифицируются в соответствии с требованиями к безопасности криптографических модулей FIPS 140-2 (Security Requirements for Cryptographic Modules). Этот довольно старый документ был опубликован в 2001 году и, разумеется, не учитывает ряд выявленных с тех пор атак. К счастью, в 2019 году он был заменен более современной версией — FIPS 140-3, которая опирается на два международных стандарта:

- *ISO/IEC 19790:2012*, который определяет четыре уровня безопасности для этих аппаратных модулей. HSM 1-го уровня не обеспечивают никакой защиты от физических атак (их можно рассматривать в качестве чисто программных реализаций), тогда как HSM 3-го уровня стирают хранящиеся в них секреты при обнаружении вторжения;
- *ISO 24759:2017*, который определяет способы тестирования HSM на предмет соответствия требованиям, предъявляемым к HSM-продуктам.

К сожалению, эти два стандарта не бесплатные. За ознакомление с ними вам придется заплатить.

США, Канада и некоторые другие страны предписывают определенным отраслям, например банковской, использовать устройства, сертифицированные в соответствии с уровнями безопасности FIPS 140. Этим же рекомендациям следуют многие компании по всему миру.

ПРИМЕЧАНИЕ

Под стиранием секретов подразумевается практика, называемая *обнулением* (zeroization). В отличие от HSM 3-го уровня HSM 4-го уровня могут перезаписывать секретные данные многократно даже в случае отключения питания благодаря внутренним резервным батареям.

Обычно можно встретить HSM в виде внешнего устройства (рис. 13.2), подключаемого к корпоративному серверу в центре обработки данных, в виде PCIe-карты,

подключаемой к материнской плате сервера, или даже в виде небольших электронных ключей, напоминающих аппаратные токены безопасности. Их можно подсоединять к оборудованию через USB-устройства (если вас не смущает низкая производительность). Некоторые из этих HSM можно администрировать с помощью смарт-карт с целью установки приложений, резервного копирования секретных ключей и т. д.

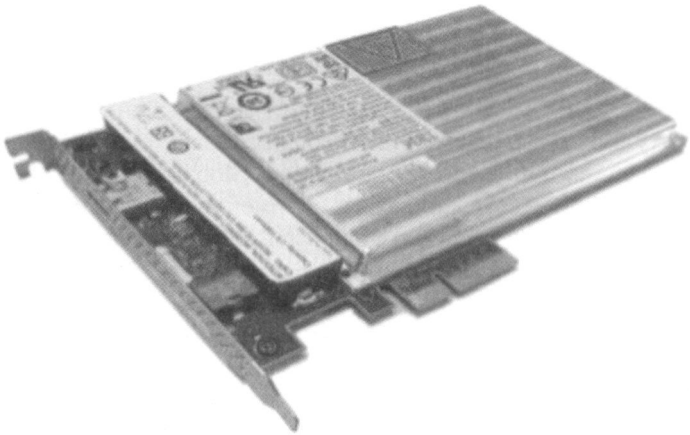


Рис. 13.2. Аппаратный модуль безопасности IBM 4767 в виде PCI-карты (изображение взято из «Википедии», http://mng.bz/XrAG)

В некоторых отраслях HSM используются особенно широко. Например, каждый раз, когда вы вводите свой PIN-код в банкомате, он проверяется именно с помощью HSM. При подключении к веб-сайту по протоколу HTTPS в качестве корня доверия выступает центр сертификации, который хранит свой закрытый ключ в HSM, а TLS-соединение может быть прервано этим модулем. Если в вашем телефоне установлена ОС Android или iPhone, имейте в виду, что компании Google и Apple, скорее всего, хранят его резервную копию в своих HSM-системах. Этот случай интересен тем, что в нем модель угроз является обратной: пользователь не доверяет облаку свои данные, поэтому поставщик облачных услуг утверждает, что его сервис не видит зашифрованную резервную копию данных пользователя и не может получить доступ к ключам, примененным для их шифрования.

У модулей HSM нет стандартного интерфейса, но большинство из них реализуют по крайней мере стандарт криптографии с открытым ключом *PKCS#11* (Public Key Cryptography Standard 11) — один из старых стандартов, который был разработан компанией RSA, а с 2012 года развивается организацией OASIS. Хотя последняя версия PKCS#11 (v2.40) была выпущена в 2015-м, это лишь обновление исходного стандарта, выпущенного в 1994 году. По этой причине она предусматривает использование ряда старых криптографических алгоритмов, то есть старых способов решения задач, что может привести к появлению уязвимостей. Тем не менее этот стандарт достаточно хорош для того, чтобы применяться во множестве

ситуаций, он определяет интерфейс, который позволяет различным системам легко взаимодействовать друг с другом. Хорошей новостью является то, что в 2020 году была выпущена версия PKCS#11 v3.0, задействующая множество современных криптографических алгоритмов, таких как Curve25519, EdDSA, SHAKE и др.

Несмотря на то что реальная цель HSM-модулей заключается в том, чтобы предотвратить извлечение ключей, их безопасность иногда оставляет желать лучшего, поскольку по большей части обеспечивается их высокой стоимостью, запретом разглашения аппаратных методов защиты и сертификатами, например FIPS и Common Criteria, которые в основном сфокусированы на аппаратном аспекте. Однако в их программном обеспечении периодически выявляются разрушительные ошибки, и узнать о том, подвержен ли используемый вами HSM какой-либо из этих уязвимостей, не всегда возможно. В 2018 году Жан-Батист Бедрун и Габриэль Кампана в своем исследовании *Everybody be Cool, This is a Robbery* продемонстрировали программную атаку, позволяющую извлечь ключи из популярных HSM-модулей.

ПРИМЕЧАНИЕ

Мало того что цена одного HSM довольно высока (в зависимости от уровня безопасности она может составлять десятки тысяч долларов), в дополнение к одному HSM вам часто требуется еще один HSM, используемый для тестирования, и еще один — для резервного копирования (на случай, если первый HSM выйдет из строя и хранящиеся в нем ключи будут потеряны). Все это может вылиться в круглую сумму!

Еще одной проблемой всех этих решений является то, что, хотя вы можете предотвратить доступ большинства злоумышленников к вашим секретным ключам, вы не можете помешать им скомпрометировать систему и осуществлять вызовы HSM (если только в этом модуле не предусмотрена логика, требующая наличия нескольких подписей или порогового количества смарт-карт для его применения). Однако в большинстве случаев единственная услуга, предоставляемая HSM, — это предотвращение незаметного похищения злоумышленником секретных данных для их последующего использования. При интеграции таких аппаратных решений, как HSM, следует сначала разобраться со своей моделью угроз, типами атак, которые вы хотите предотвратить, и определить, не являются ли пороговые схемы вроде мультиподписи, обсуждавшейся в главе 8, более предпочтительным решением.

13.2.4. Доверенные платформенные модули — полезная стандартизация защищенных элементов

Хотя защищенные элементы и модули HSM доказали свою полезность, спектр случаев их применения весьма ограничен, а процесс написания предназначенных для них приложений довольно утомителен. По этой причине *Trusted Computing Group* (TCG), еще одна некоммерческая организация, созданная участниками отрасли, предложила готовое альтернативное решение, предназначенное как для персональных, так и для корпоративных компьютеров и известное как *доверенный платформенный модуль* (Trusted Platform Module, TPM).

TPM — это не чип, а стандарт (TPM 2.0), который при желании может внедрить любой производитель. Модуль, соответствующий стандарту TPM 2.0, представляет собой защищенный и устойчивый к взлому микроконтроллер, оснащенный аппаратным генератором случайных чисел, защищенной памятью для хранения секретных данных и способный выполнять криптографические операции. Это описание может показаться знакомым, и действительно, TPM довольно часто реализуются путем перекомпоновки защищенных элементов. Обычно TPM припаивается или подключается непосредственно к материнской плате корпоративных серверов, ноутбуков и настольных компьютеров (рис. 13.3).

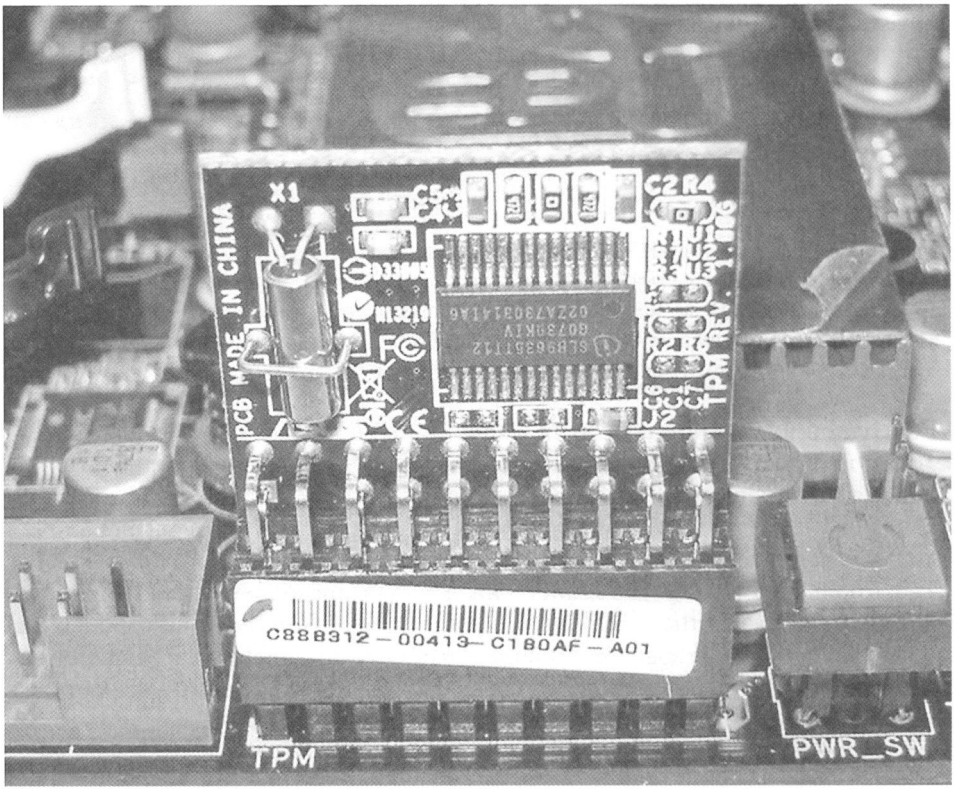


Рис. 13.3. Микросхема, соответствующая стандарту TPM 2.0 и подключенная к материнской плате. К этой микросхеме могут обращаться как компоненты системной платы, так и пользовательские приложения, запущенные в ОС компьютера (изображение взято из «Википедии», http://mng.bz/Q2je)

В отличие от смарт-карт и защищенных элементов, модуль TPM не способен выполнять произвольный код. Вместо этого он предоставляет четко определенный интерфейс, которым может воспользоваться более мощная система. Как правило, TPM стоят довольно дешево, поэтому ими оснащены многие современные портативные компьютеры.

А теперь о плохом: канал связи между модулем TPM и процессором обычно представляет собой шинный интерфейс, который злоумышленник может легко перехватить, если ему удастся украсть устройство или получить временный физический доступ к нему. Несмотря на то что многие TPM обеспечивают высокий уровень защиты от физических атак, из-за своего в некотором смысле открытого канала связи они в основном применяются для защиты от программных атак.

Для решения этих проблем были разработаны TPM-подобные чипы, интегрируемые непосредственно в основной процессор. Их примеры — Secure Enclave компании Apple и Pluton компании Microsoft. К сожалению, ни один из этих чипов безопасности не соответствует какому бы то ни было стандарту, что усложняет или даже делает невозможным применение их функциональности пользовательскими приложениями. Чтобы получить представление о способностях таких аппаратных чипов безопасности, как TPM, рассмотрим несколько примеров.

Простейший вариант применения модулей TPM — защита данных. Для защиты ключей достаточно сгенерировать их в защищенном чипе и запретить извлечение. Если вам нужны ключи, попросите чип выполнить криптографические операции. Чтобы защитить данные, зашифруйте их. Данная концепция называется *пофайловым шифрованием* (file-based encryption, FBE), если вы шифруете отдельные файлы, и *полнодисковым шифрованием* (full-disk encryption, FDE), если речь идет обо всем диске. «Полнодисковое шифрование» звучит гораздо лучше, поскольку использует подход «все или ничего». Именно его реализует большинство ноутбуков и настольных компьютеров. Однако на практике FDE-шифрование оказывается не столь уж хорошим решением, так как не учитывает то, как люди используют свои устройства. Мы часто не выключаем, а просто блокируем их, позволяя фоновым функциям продолжать работу. При этом заблокированные компьютеры держат наготове ключ шифрования данных (DEK). (Вспомните об этом, когда в следующий раз отправитесь в туалет в кофейне, оставив заблокированный компьютер без присмотра.) Современные телефоны обеспечивают больший уровень безопасности, шифруя различные типы файлов в зависимости от того, находится телефон в заблокированном или выключенном состоянии.

ПРИМЕЧАНИЕ

На практике реализация FDE и FBE имеет множество проблем. В 2019 году Мейер и Гастел, авторы статьи Self-encrypting deception: Weaknesses in the encryption of solid state drives (SSDs), показали, что некоторые производители SSD используют совершенно небезопасные решения. В 2021 году авторы статьи Data Security on Mobile Devices: Current State of the Art, Open Problems, and Proposed Solutions Зинкус и другие обнаружили множество проблем с шифрованием дисков мобильных телефонов.

Разумеется, перед расшифровкой данных пользователь должен пройти аутентификацию. Для этого часто применяется PIN-код или пароль. Однако только их одних недостаточно, поскольку они уязвимы к простым атакам методом грубой силы (особенно это касается четырех- или шестизначных PIN-кодов). Большинство решений пытается привязать DEK как к учетным данным пользователя, так и к симметричному ключу, хранящемуся в анклаве.

Однако производитель чипов не может жестко закодировать один и тот же ключ во всех выпускаемых им устройствах, так как это сделает их уязвимыми к атакам наподобие DUHK (https://duhkattack.com), затронувшей тысячи устройств, в которые был жестко кодирован один и тот же секрет. При этом компрометация одного устройства приводит к компрометации всех остальных! Решением этой проблемы является отдельный ключ для каждого устройства, который либо встраивается в чип на этапе производства, либо создается самим чипом с помощью аппаратных компонентов, называемых *физически неклонируемыми функциями*. Например, каждый чип Apple Secure Enclave имеет уникальный идентификатор (UID), каждый TPM-модуль — уникальный ключ подтверждения и ключ удостоверения подлинности и т. д. Чтобы защититься от атак методом грубой силы, чип Secure Enclave Apple дополняет UID-ключ и PIN-код пользователя функцией формирования ключа на основе пароля (о которой говорилось в главе 2) для генерации ключа шифрования данных (DEK). Правда, для того, чтобы пользователи могли быстро изменить свой PIN-код, DEK выводится не напрямую, а шифруется ключом шифрования ключа (KEK).

Еще один пример — *безопасная загрузка*. Процесс загрузки компьютера состоит из нескольких этапов. И одной из проблем, с которой сталкиваются пользователи, являются вирусы и вредоносные программы, которые в случае заражения процесса загрузки запускают вредоносную ОС.

Чтобы защитить целостность процесса загрузки, модули TPM и интегрированные защищенные чипы обеспечивают корень доверия, то есть то, чему мы доверяем на 100 % и что позволяет нам в дальнейшем доверять другим вещам. В качестве корня доверия обычно выступает память, доступная только для чтения, которую нельзя перезаписать, также называемая *одноразовой программируемой памятью*, поскольку данные записываются в нее на этапе производства и впоследствии не могут быть изменены. Например, при включении относительно нового устройства Apple первым делом запускается загрузочное ПЗУ, расположенное внутри чипа Secure Enclave. Это ПЗУ очень маленькое, поэтому обычно оно выполняет только следующие действия.

1. Подготавливает защищенную память и загружает в нее следующую программу для запуска (обычно какой-нибудь другой загрузчик).
2. Хеширует эту программу и проверяет ее подпись с помощью открытого ключа, жестко закодированного в ПЗУ.
3. Выполняет программу.

Следующий загрузчик делает то же самое, и так продолжается вплоть до запуска ОС. Кстати, именно поэтому обновления ОС, не подписанные Apple, не могут быть установлены на ваш телефон.

Модули TPM и интегрированные TPM-подобные чипы представляют собой интересную разработку, которая позволила существенно повысить безопасность наших устройств за последнее время. По мере снижения их стоимости и разработки стандартов все больше устройств смогут воспользоваться преимуществами, которые они обеспечивают.

13.2.5. Конфиденциальные вычисления с помощью доверенной среды выполнения

Смарт-карты, защищенные элементы, HSM и TPM представляют собой автономные чипы или модули. Они оснащены собственным процессором, памятью, TRNG и т. д., а другие компоненты могут взаимодействовать с ними с помощью проводов или радиочастот, если речь идет о микросхемах с поддержкой технологии NFC. TPM-подобные чипы (Pluton компании Microsoft и Secure Enclave компании Apple) тоже автономные, хоть и тесно связаны с главным процессором внутри системы на кристалле (SoC). В этом разделе мы поговорим о следующем логическом шаге в таксономии аппаратных средств безопасности, а именно об *интегрированной безопасности* (integrated security) — аппаратных средствах защиты, применяемых на уровне главного процессора.

Процессоры с интегрированным чипом безопасности создают *доверенную среду выполнения* (trusted execution environment, TEE) пользовательского кода путем расширения набора инструкций процессора, благодаря которому программы могут выполняться в отдельной защищенной среде. Разделение между этой безопасной и привычной *функционально богатой средой выполнения* осуществляется с помощью аппаратных средств. В итоге на современных процессорах одновременно работают как обычная, так и защищенная ОС. Обе имеют собственный набор регистров, но делят между собой большую часть остальной архитектуры процессора. Благодаря логике, навязываемой процессором, из обычной среды невозможно получить доступ к данным, находящимся в защищенной среде. Например, процессор обычно разделяет свою память, выделяя одну ее часть исключительно для использования TEE. Поскольку TEE реализуется непосредственно в главном процессоре, это говорит не только о том, что она является более быстрым и дешевым продуктом по сравнению с модулем TPM или защищенным элементом, но и о том, что многие современные процессоры предоставляют ее бесплатно.

TEE, как и все прочие аппаратные решения, представляет собой концепцию, разработанную разными производителями независимо друг от друга, и стандарт, созданный Global Platform. Наиболее известные TEE — это Software Guard Extensions (SGX) от Intel и TrustZone от ARM.

Чтобы понять, для чего нужны TEE, рассмотрим пример. За последние несколько лет сформировалась новая, облачная парадигма, в рамках которой крупные компании задействуют свои серверы для размещения данных пользователей. У компании Amazon для этого есть сервис AWS, у Google — GCP, а у Microsoft — Azure. Иначе говоря, для решения задач люди все чаще используют не свое, а чужое оборудование. Это создает определенные проблемы в тех ситуациях, когда важна конфиденциальность. Именно их призваны исправить так называемые *конфиденциальные вычисления* — решения, позволяющие запускать клиентский код, не имея возможности просматривать его или изменять его поведение. Судя по всему, сейчас SGX применяется именно с этой целью — позволить клиентам запускать код, который серверы не могут увидеть или изменить.

В связи с этим возникает интересная проблема: как можно удостовериться в том, что ответ на запрос был получен именно от SGX, а не от какого-то самозванца? Для ее решения предназначена *аттестация*, которая бывает двух видов.

- *Локальная аттестация.* Два анклава, работающие на одной платформе, должны взаимодействовать и доказывать друг другу то, что они безопасны.
- *Удаленная аттестация.* Клиент, направивший запрос удаленному анклаву, должен убедиться в легитимности анклава, от которого пришел ответ.

На этапе производства каждый чип SGX снабжается уникальной парой ключей (root sealing keys). Затем открытый ключ подписывается центром сертификации Intel. Если отбросить предположение о безопасности аппаратного обеспечения, то первое допущение будет заключаться в том, что Intel правильно подписывает открытые ключи только безопасных чипов SGX. Учитывая это, вы можете получить от ЦС Intel подписанное удостоверение, подтверждающее, что вы взаимодействуете с настоящим анклавом SGX и что он выполняет определенный код.

Первостепенная цель TEE заключается в предотвращении *атак на программное обеспечение*, и эта заявленная программная безопасность кажется весьма привлекательной идеей. Однако на практике выполнение кода на одном чипе бывает трудно разделить из-за чрезвычайной сложности современных процессоров и их динамических состояний. Об этом свидетельствуют многочисленные программные атаки на SGX и TrustZone (https://foreshadowattack.eu, https://mdsattacks.com, https://plundervolt.com и https://sgaxe.com).

Концепция TEE обеспечивает определенную устойчивость к физическим атакам, поскольку то, что происходит на этом микроскопическом уровне, сложно проанализировать без дорогостоящего оборудования. Однако защититься от мотивированного злоумышленника может быть сложнее.

13.3. КАКОЕ РЕШЕНИЕ ПОДХОДИТ ИМЕННО МНЕ?

В этой главе вы узнали о множестве аппаратных продуктов. Далее приведен их перечень, проиллюстрированный на рис. 13.4.

- *Смарт-карты — это микрокомпьютеры, которые включаются внешним устройством, например платежным терминалом.* На них можно выполнять небольшие пользовательские Java-подобные приложения. Широко распространенный пример смарт-карты — банковская карта.
- *Защищенные элементы представляют собой обобщение смарт-карт, которое опирается на набор стандартов ассоциации Global Platform.* Пример защищенного элемента — SIM-карта.
- *HSM (аппаратные модули безопасности) можно рассматривать в качестве более крупных защищенных элементов, подключаемых к корпоративным сер-*

верам. Они работают быстрее и гибче и в основном используются в центрах обработки данных для хранения секретных ключей, что делает атаки на эти ключи более заметными.

- *TPM (доверенные платформенные модули) — это переупакованные защищенные элементы, подключаемые к материнским платам персональных и корпоративных компьютеров*. Они применяют стандартный API, разработанный организацией Trusted Computing Group, который предоставляет операционным системам и конечным пользователям различные функциональные возможности.
- *Процессоры безопасности — это TPM-подобные чипы, которые встраиваются очень близко к главному процессору и не поддаются программированию*. Они не соответствуют никаким стандартам, и разные производители используют различные технологии.
- *TEE (доверенные среды выполнения), такие как TrustZone и SGX, можно рассматривать в качестве программируемых защищенных элементов, реализованных в рамках набора инструкций процессора*. Они представляют собой более быстрый и дешевый продукт, который в основном обеспечивает защиту от программных атак. Большинство современных процессоров поставляется с TEE и предусматривает различные уровни защиты от аппаратных атак.



Рис. 13.4. Внешний вид аппаратных решений, описанных в этой главе

Чтобы выбрать оптимальное для себя решение, задайте себе следующие вопросы.

- *«Какой формфактор мне подойдет?»* Например, потребность в защищенном элементе для небольшого устройства определит то, какие решения вы не сможете использовать.

- *«Какая скорость мне нужна?»* Если вашим приложениям необходимо выполнять большое количество криптографических операций в секунду, то выбор решений, скорее всего, будет ограничен такими вариантами, как HSM и TEE.
- *«Какой уровень безопасности мне необходим?»* Об уровнях безопасности программного или аппаратного обеспечения можно судить по сертификатам и заявлениям производителей.

Помните о том, что ни одно из аппаратных решений не является панацеей — с их помощью вы всего лишь увеличиваете стоимость реализации атаки. Против изощренного злоумышленника все эти решения практически бесполезны. Проектируйте систему так, чтобы компрометация одного устройства не приводила к компрометации остальных.

13.4. КРИПТОГРАФИЯ, УСТОЙЧИВАЯ К УТЕЧКАМ, ИЛИ КАК СМЯГЧИТЬ АТАКИ ПО СТОРОННИМ КАНАЛАМ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Ранее мы говорили о том, как аппаратные средства пытаются предотвратить прямое наблюдение и извлечение секретных ключей, однако возможности аппаратных средств в этом отношении довольно ограничены. В конце концов, программное обеспечение может выдать ключ, несмотря на всю аппаратную защиту. Программа может сделать это напрямую (как в случае с бэкдором) или косвенным способом, передав достаточно информации для того, чтобы злоумышленник смог восстановить ключ. В последнем случае речь идет о так называемых *сторонних каналах*, уязвимости которых чаще всего оказываются результатом непреднамеренных ошибок (по крайней мере, хочется на это надеяться).

Я упоминал об атаках по времени в главе 3, где говорилось о том, что аутентификационные теги должны сравниваться за постоянное время, в противном случае злоумышленники могут вычислить правильный тег, отправив вам множество неправильных и измерив время ожидания ответа. Атаки по времени обычно воспринимаются всерьез во всех областях реальной криптографии, поскольку они могут быть реализованы удаленно через сеть, в отличие от атак по физическим сторонним каналам.

Наиболее важный и известный сторонний канал — это *энергопотребление*, о котором я упоминал ранее в этой главе. В 1998 году Кохер, Джаффе и Джун обнаружили возможность осуществления так называемой *дифференциальной атаки по энергопотреблению* (differential power analysis, DPA), когда поняли, что, подключив осциллограф к устройству, можно наблюдать за тем, как изменяется уровень потребляемой устройством мощности в процессе шифрования известных открытых текстов. Эти изменения явно зависят от битов используемого ключа и того факта, что операции наподобие XOR потребляют больше или меньше энергии в зависимости от того, были установлены биты операнда или нет. Это наблюдение легло в основу атаки, направленной на извлечение ключа (key-extraction attack).

Данную концепцию можно проиллюстрировать на примере *простой атаки по энергопотреблению* (simple power analysis, SPA). В идеальной ситуации и при отсутствии аппаратных и программных средств защиты от такого рода атак бывает достаточно измерить и проанализировать энергопотребление одной криптографической операции, выполняемой с помощью секретного ключа (рис. 13.5).

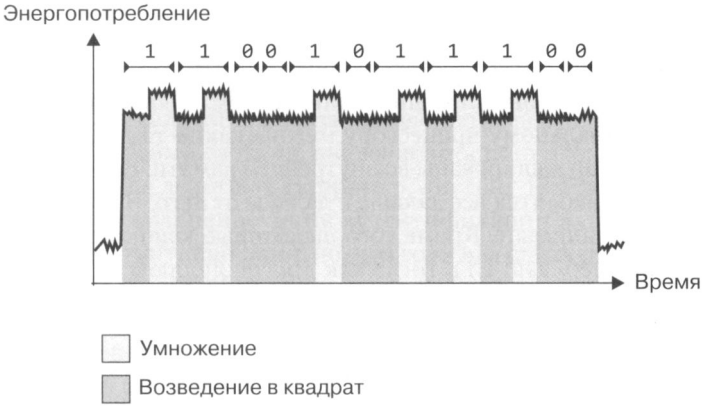


Рис. 13.5. Некоторые криптографические алгоритмы выдают через энергопотребление так много информации, что простой анализ графика, отражающего динамику потребления мощности во времени, может позволить злоумышленнику восстановить закрытый ключ алгоритма. Например, здесь представлен график энергопотребления алгоритма RSA при возведении сообщения в степень с использованием секретной экспоненты (см. главу 6). В RSA возведение в степень реализуется с помощью алгоритма возведения в квадрат и умножения, который перебирает биты секретной экспоненты. К каждому биту применяется операция возведения в квадрат, а затем, если этот бит установлен, и операция умножения. Здесь очевидно, что операция умножения потребляет больше энергии, что делает график энергопотребления весьма информативным

Энергопотребление — не единственный физический сторонний канал. При некоторых атаках анализируются электромагнитные излучения, вибрации и даже звук, издаваемый аппаратурой. Несмотря на то что данная глава посвящена аппаратному обеспечению, я хочу упомянуть еще о двух важных нефизических сторонних каналах, средства защиты которых должны предусматриваться во многих реальных криптографических приложениях.

Во-первых, к утечке критически важной информации иногда могут приводить возвращаемые ошибки. Например, в 2018 году разработчики атаки ROBOT нашли способ осуществления атаки Блейхенбахера (упомянута в главе 6) на ряде серверов, которые реализовали стандарт RSA PKCS#1 v1.5 в протоколе TLS (рассмотрен в главе 9). Атака Блейхенбахера работает только в том случае, если вы можете определить, какие блоки дополнения содержит RSA-шифротекст — корректные или некорректные. Для защиты от этой атаки безопасные реализации проверяют блоки дополнения за постоянное время и не возвращают раньше срока, если

обнаруживают недействительный блок. Например, если блок дополнения полезной нагрузки RSA некорректный, то при обмене RSA-ключами в рамках TLS-протокола сервер должен представить ответ, имитирующий успешное завершение рукопожатия. Однако если по окончании этой проверки реализация возвратит клиенту другую ошибку (основанную на действительности блока дополнения), значит, все было напрасно.

Во-вторых, обращение к памяти может занимать больше или меньше времени в зависимости от того, обращались к соответствующим данным ранее или нет. Это связано с многочисленными уровнями кэширования, существующими в компьютере. Например, если центральному процессору нужны какие-то данные, он сначала проверяет, не были ли они кэшированы во внутренней памяти. Если нет, то обращается к кэшам, которые находятся все дальше и дальше от него. Чем дальше кэш, тем больше времени это занимает. Кроме того, некоторые кэши относятся только к одному ядру (например, кэш L1), тогда как другие являются общими для нескольких ядер многоядерного процессора (например, кэш L3, оперативная память, диск).

Атаки на кэш основаны на том факте, что вредоносная программа может быть запущена на той же машине и может использовать ту же криптографическую библиотеку, что и важная криптографическая программа. Например, многие облачные сервисы размещают различные виртуальные серверы на одной машине, и многие серверы задействуют библиотеку OpenSSL для выполнения криптографических операций или выдачи TLS-страниц. Вредоносные программы находят способы удалить части библиотеки, загруженные в кэш, который они используют совместно с процессом жертвы, а затем периодически измеряют время, затрачиваемое на повторное чтение некоторых частей библиотеки. Если это занимает много времени, значит, жертва не выполняла данную часть программы, если же происходит быстро, значит, жертва обращалась к этой части программы, поместив ее в кэш, чтобы не пришлось снова считывать ее из удаленного кэша или, что еще хуже, с диска. В результате получается кривая, напоминающая график энергопотребления, которую можно использовать аналогичным образом!

Итак, мы кратко ознакомились с темой атак на сторонние каналы. Если она вас заинтересовала, можете обратиться к другим, более исчерпывающим ресурсам. В этом разделе я хочу рассказать только о программных средствах защиты, которые криптографические реализации могут и должны применять для предотвращения атак по сторонним каналам в целом. Данная область исследований называется *криптографией, устойчивой к утечкам* (leakage-resilient cryptography), поскольку цель криптографа в данном случае — предотвращение попадания секретных данных в руки третьих лиц.

Защита от физических атак — это бесконечная борьба, что объясняет, почему многие из соответствующих средств защиты являются проприетарными и не предполагают раскрытия своих секретов. Информация, изложенная в разделе, разумеется, не исчерпывающая, она лишь призвана дать вам представление о том, что делают криптографы для защиты от атак по сторонним каналам.

13.4.1. Программирование операций, выполняемых за постоянное время

Первой линией защиты для любой криптографической программы является реализация выполнения ее криптографически чувствительных частей (любых вычислений, связанных с секретными данными) за постоянное время. Это позволяет защититься не только от атак по времени, но и от многих других классов атак, в частности атак на кэш и простой атаки по энергопотреблению.

Чтобы реализовать некую операцию за постоянное время, не допускайте *ветвления*. Другими словами, вне зависимости от входных данных всегда выполняйте одно и то же действие. Например, листинг 13.1 демонстрирует, как язык Golang реализует сравнение тегов аутентификации алгоритма HMAC за постоянное время. Интуитивно понятно, что если два байта равны, то результат применения к ним операции XOR будет равен 0. Если это свойство проверяется для каждой пары сравниваемых байтов, то применение к ним операции OR также приведет к получению значения 0 (в противном случае оно будет отличаться от нуля). Если вы впервые сталкиваетесь с подобными трюками, то этот код может показаться несколько сложным для восприятия.

Листинг 13.1. Сравнение двух байтовых массивов за постоянное время в Golang

```
func ConstantTimeCompare(x, y []byte) byte {
    if len(x) != len(y) {
        return 0
    }

    var v byte
    for i := 0; i < len(x); i++ {
        v |= x[i] ^ y[i]
    }

    return v
}
```

Нет смысла сравнивать две строки за постоянное время, если они имеют разную длину

Здесь происходит волшебство. Цикл, выполняющий операцию OR, накапливает результаты операции XOR каждого байта в переменной v

Возвращает 0 только в том случае, если v равна 0, в противном случае возвращает ненулевое значение

Для сравнения тегов аутентификации MAC можно остановиться на этом этапе, чтобы проверить, равен результат 0 или нет, путем ветвления (с помощью условного выражения типа `if`). Другой интересный пример — *скалярное произведение*, используемое в эллиптической криптографии, которое, как было сказано в главе 5, предполагает сложение точки с самой собой x раз, где x — это то, что мы называем скаляром. Процесс может быть несколько медленным, но для его ускорения существуют «умные» алгоритмы. Один из самых популярных — *алгоритм Монтгомери* (Montgomery ladder), который практически идентичен алгоритму возведения в квадрат и умножения RSA, который я упоминал ранее, но относится к другой группе.

Алгоритм Монтгомери предполагает чередование операций сложения двух точек и удвоения точки (ее сложения с самой собой). Алгоритм возведения в квадрат и умножения RSA и алгоритм Монтгомери используют простой способ защиты от атак по времени: они не допускают ветвления и всегда выполняют обе операции.

Именно поэтому алгоритм RSA, выполняющий возведение в степень за постоянное время, обычно называют методом *всегда выполняющихся возведения в квадрат и умножения* (square and multiply always).

ПРИМЕЧАНИЕ

В главе 7 я упомянул о том, что схемы цифровой подписи могут допускать различные ошибки и что существуют атаки, направленные на восстановление ключей и применяемые против реализаций, допускающих утечку нескольких байтов используемых nonce (примером такой схемы является алгоритм ECDSA). Именно это произошло в ходе атак Minerva и TPM-Fail, реализованных примерно в одно и то же время. Обе эти атаки эксплуатировали уязвимость ряда устройств, обусловленную вариацией в длительности выполнения операций цифровой подписи.

На практике защититься от атак по времени бывает непросто, поскольку не всегда ясно, выполняются ли инструкции процессора, связанные с умножением или условным перемещением, за постоянное время. Кроме того, не всегда понятно, как компилятор будет компилировать высокоуровневый код при использовании других флагов компиляции. По этой причине для получения большей уверенности в написанном коде, выполняемом за постоянное время, иногда проводится ручная проверка сгенерированного ассемблера. Для анализа такого кода существуют различные инструменты (в частности, ducdect, ct-verif и SideTrail), но на практике они применяются довольно редко.

13.4.2. Не используйте секрет! Маскировка и ослепление

Еще один распространенный способ, позволяющий помешать злоумышленникам или хотя бы сбить их с толку, заключается в добавлении дополнительных уровней косвенности в любую операцию с секретными данными. Одна из таких техник называется *ослеплением*, которое часто оказывается возможным благодаря арифметической структуре алгоритмов криптографии с открытым ключом. В главе 11 было показано, как ослепление используется в забывчивых алгоритмах, таких как алгоритмы обмена ключами с парольной аутентификацией, и мы можем задействовать ослепление таким же образом, если хотим сделать забывчивой стороной злоумышленника, наблюдающего утечку данных из наших вычислений. Рассмотрим в качестве примера алгоритм RSA.

Напомню, этот алгоритм расшифровывает шифротекст c путем возведения его в степень с использованием секретной экспоненты d, что отменяет применение открытой экспоненты e, с помощью которой вычислялся шифротекст ($m^e \bmod N$). Если не помните деталей, обратитесь к главе 6. Один из способов добавления косвенности предполагает выполнение операции расшифровки над значением, которое не является шифротекстом, известным злоумышленнику. Этот метод называется *ослеплением базы* (base blinding) и сводится к выполнению следующих шагов.

1. Генерация случайного фактора ослепления r.
2. Вычисление значения *сообщение* = (*шифротекст* $\times r^e$)$^d \bmod N$.
3. Отмена ослепления результата путем вычисления значения *реальное_сообщение* = *сообщение* $\times r^{-1} \bmod N$, где r^{-1} — это величина, обратная r.

Данный метод позволяет ослепить значение, используемое наряду с секретом, однако мы можем ослепить и сам секрет. Например, нахождение скалярного произведения точек эллиптической кривой обычно выполняется с применением секретного скаляра. Но, поскольку вычисления происходят в циклической группе, прибавление к этому секрету числа, кратного порядку, не приводит к изменению результата вычислений. Эта техника называется *ослеплением скаляра* (scalar blinding) и выполняется следующим образом.

1. Генерация случайного значения k_1.
2. Вычисление скаляра $k_2 = d + k_1 \times порядок$, где d — исходный секретный скаляр, а *порядок* — его порядок.
3. Чтобы вычислить $Q = P$, вычисляем $Q = [k_2]P$, что дает ту же точку.

Все эти методы доказали свою эффективность и часто применяются в сочетании с другими программными и аппаратными средствами защиты. В симметричной криптографии используется еще одна похожая техника, называемая *маскировкой*.

Маскировка предполагает преобразование входных данных (открытого текста или шифротекста) перед их передачей алгоритму, например перед выполнением операции XOR над входными данными с помощью случайного значения. Затем отменяется маскировка выхода для получения итогового правильного результата. Подобная маскировка промежуточных состояний обеспечивает некоторую декорреляцию между криптографическими вычислениями и входными данными и значительно затрудняет реализацию атак по сторонним каналам. Алгоритм должен знать об этой маскировке, чтобы правильно выполнять внутренние операции, придерживаясь при этом исходного корректного поведения.

13.4.3. А как насчет атак по ошибкам вычислений?

Ранее я уже рассказывал об *атаках по ошибкам вычислений* — более агрессивном типе атак по сторонним каналам, которые изменяют процесс выполнения алгоритма, вызывая сбои. Они могут вызываться множеством физических способов, например нагревом системы или даже обстрелом лазером определенных точек целевого чипа.

Сбои могут быть спровоцированы и с помощью программного обеспечения. Примеры — атаки Plundervolt и V0LTpwn, в основе которых лежит изменение напряжения процессора, вызывающее естественные сбои в его работе. Еще одной печально известной атакой является rowhammer, позволяющая изменить значение близлежащих битов путем многократного обращения к динамической памяти. Подобные атаки труднореализуемые, но чрезвычайно мощные. В криптографии вычисление некорректного результата иногда может привести к утечке ключа. Например, именно это происходит с подписями RSA, реализованными с помощью некоторых специфических оптимизаций.

Хотя полностью защититься от этих атак невозможно, существуют приемы, позволяющие повысить сложность их успешной реализации. Примером является многократное вычисление результатов одной и той же операции и их сравнение на предмет совпадения перед выдачей или верификация результата перед его выдачей. В частности, подпись можно проверить с помощью открытого ключа перед ее возвращением.

Атаки по ошибкам вычислений могут иметь серьезные последствия и для генераторов случайных чисел. Одно из простых решений — применение алгоритмов, не использующих новое случайное значение при каждом запуске. Например, в главе 7 вы узнали об алгоритме цифровой подписи EdDSA, который не требует нового случайного значения для создания очередной подписи, в отличие от алгоритма ECDSA.

В целом ни одна из этих техник не является абсолютно надежной. Криптография в средах с высокой степенью враждебности всегда сводится к повышению издержек для злоумышленников.

РЕЗЮМЕ

- Сейчас опасность представляет не только то, что злоумышленник может перехватить сообщения, передаваемые по проводам, но и то, что он может украсть устройство, на котором работает ваша криптографическая система, или каким-то образом повлиять на него. Устройства, относящиеся к сфере Интернета вещей (IoT), часто подвергаются подобным угрозам и по умолчанию беззащитны перед изощренными злоумышленниками. В последнее время облачные сервисы также начали учитываться в модели угроз их пользователям.
- Аппаратные средства могут помочь защитить криптографические приложения и их секреты в средах с высокой враждебностью. Один из способов заключается в том, чтобы снабдить устройство устойчивым к взлому чипом для хранения секретных данных и выполнения криптографических операций. Если такое устройство попадет в руки злоумышленника, ему будет непросто извлечь ключи или изменить поведение чипа.
- Считается, что для защиты криптографических систем в средах с высокой враждебностью необходимо сочетать различные программные и аппаратные методы. Однако аппаратные средства защиты не панацея — такая глубоко эшелонированная защита всего лишь сдерживает злоумышленника и увеличивает стоимость реализации атаки. Противники, имеющие неограниченные время и деньги, всегда найдут способ взломать ваше оборудование.
- Уменьшение потенциальных последствий атаки тоже может отпугнуть злоумышленников. Для этого необходимо как следует продумать дизайн системы, например сделать так, чтобы компрометация одного устройства не приводила к компрометации всех остальных.
- Существует множество аппаратных решений, наиболее популярны следующие.
  - Смарт-карты. Они стали одними из первых защищенных микроконтроллеров, которые можно было использовать в качестве микрокомпьютера для хранения секретов и выполнения криптографических операций. Предполагается, что в них задействуется ряд техник, препятствующих физическим атакам. Обобщением концепции смарт-карты стал так называемый защищенный элемент. Данный термин используется по-разному в различных областях, но, как правило, обозначает смарт-карту, которая может применяться в качестве сопроцессора в более крупной системе, уже имеющей главный процессор.

  - Аппаратные модули безопасности (HSM). Их часто называют подключаемыми картами, которые действуют как защищенные элементы. Они не имеют стандартного интерфейса, но обычно реализуют стандарт PKCS#11 для криптографических операций. HSM могут иметь различные уровни безопасности и сертифицироваться в соответствии со стандартом NIST (FIPS 140-3).
  - Доверенные платформенные модули (TPM). Они похожи на защищенные элементы с определенным интерфейсом, соответствующим стандарту TPM 2.0. Обычно TPM подключаются к материнской плате ноутбука или сервера.
  - Доверенная среда выполнения (TEE). Это способ разделения среды выполнения на безопасную и потенциально небезопасную. Обычно TEE реализуются в виде расширения набора инструкций процессора.
- Для защиты криптографических операций в средах с высокой враждебностью одних аппаратных средств недостаточно, поскольку программные и аппаратные атаки по сторонним каналам могут эксплуатировать различные источники утечки данных. Примерами таких источников являются время выполнения, энергопотребление, электромагнитные излучения оборудования и т. д. Для защиты от атак по сторонним каналам криптографические алгоритмы используют программные средства защиты.
  - Серьезные криптографические реализации основаны на алгоритмах, выполняемых за постоянное время, и избегают любых ветвлений, а также обращений к памяти, зависящих от секретных данных.
  - Методы защиты, основанные на ослеплении и маскировке, позволяют обеспечить декорреляцию между криптографическими операциями и секретом или данными, над которыми они выполняются.
  - От атак по ошибкам вычислений защититься сложнее. К методам защиты относится многократное вычисление результата операции, его сравнение и проверка (например, проверка подписи с помощью открытого ключа) перед выдачей.
- Усиление криптографической защиты во враждебных условиях — это бесконечная борьба. Используйте комбинацию программных и аппаратных средств защиты, чтобы увеличить стоимость успешной реализации атаки для злоумышленника до приемлемого уровня риска. Вам также следует уменьшить потенциальные последствия атаки, применяя уникальные ключи для каждого устройства и, возможно, даже для каждой криптографической операции.

11 Постквантовая криптография

В этой главе

- ✓ Квантовые компьютеры и их влияние на криптографию.
- ✓ Постквантовая криптография для защиты от квантовых компьютеров.
- ✓ Постквантовые алгоритмы сегодняшнего и завтрашнего дня.

«Квантовые компьютеры способны разрушить криптографические системы», — сказал в 1994 году Питер Шор, профессор математики из Массачусетского технологического института, придумав незадолго до этого новый алгоритм. Его открытие было связано с возможностью эффективной факторизации целых чисел, что обещало разрушить криптографические алгоритмы вроде RSA в случае изобретения квантовых компьютеров. В то время квантовый компьютер представлял собой лишь теоретическую концепцию нового класса компьютеров, основанных на явлениях квантовой физики. Жизнеспособность этой идеи еще предстояло доказать. В середине 2015 года Агентство национальной безопасности (АНБ) поразило всех, объявив о своих планах перехода на применение *квантово-устойчивых алгоритмов* (криптографических алгоритмов, неуязвимых для квантовых компьютеров).

> Партнерам и поставщикам, которые еще не перешли на использование алгоритмов Suite B на основе эллиптических кривых, мы рекомендуем воздержаться от значительных инвестиций в этом направлении, а вместо этого подготовиться к предстоящему переходу на применение квантово-устойчивых алгоритмов. <...> К сожалению, рост использования эллиптических кривых натолкнулся на факт последовательного прогресса в исследованиях квантовых вычислений, который дал понять, что криптография на основе эллиптических кривых не является тем долгосрочным решением, на которое многие надеялись. Учитывая это, мы были вынуждены скорректировать свою стратегию.
>
> *Агентство национальной безопасности США (Cryptography Today, 2015)*

Хотя идея *квантовых вычислений* (создания компьютера на основе физических явлений, изучаемых в области квантовой механики) не была новой, за предшествующие годы на ее реализацию было выделено огромное количество грантов. Однако, несмотря на значительные экспериментальные прорывы, никому так и не удалось продемонстрировать успешный взлом криптографических систем с помощью квантового компьютера. Знало ли АНБ что-то, чего не знали все остальные? Действительно ли квантовые компьютеры способны взломать криптографические системы? И что такое квантово-устойчивая криптография? В этой главе я постараюсь ответить на все эти вопросы.

14.1. ЧТО ТАКОЕ КВАНТОВЫЕ КОМПЬЮТЕРЫ И ПОЧЕМУ ОНИ ПУГАЮТ КРИПТОГРАФОВ

После заявления АНБ тему квантовых компьютеров стали часто освещать в новостях, поскольку многие крупные компании, такие как IBM, Google, Alibaba, Microsoft, Intel и др., начали вкладывать значительные средства в их исследование. Но что же представляют собой эти квантовые компьютеры и чем они так страшны? Чтобы это понять, следует обратиться к *квантовой механике* (также называемой *квантовой физикой*) — области физики, изучающей поведение мельчайших частиц (меньше атома). Поскольку именно явления квантовой механики лежат в основе квантовых компьютеров, с них мы и начнем.

> Одно время в газетах писали, что теорию относительности понимают лишь двенадцать человек. Я в это не верю. Возможно, до публикации ее понимал только сам автор. Но после того, как люди прочитали его статью, многие так или иначе поняли теорию относительности. И таких людей определенно было больше двенадцати. С другой стороны, я могу с уверенностью сказать, что никто не понимает квантовую механику.
>
> *Ричард Фейнман (The Character of Physical Law, MIT Press, 1965)*

14.1.1. Квантовая механика, изучение мельчайших частиц

Физики долго считали, что весь мир так же детерминирован, как криптографические генераторы псевдослучайных чисел. То есть если вы знаете устройство Вселенной и у вас есть достаточно большой компьютер, позволяющий вычислить ее «функцию», то, имея лишь *зерно* (информацию, содержащуюся во Вселенной в момент Большого взрыва), вы сможете предсказать любое событие. В том числе то, что спустя всего 13,7 млрд лет после возникновения Вселенной вы будете читать эти строки. В таком мире нет места случайности. Каждое ваше решение предопределено прошлыми событиями, в том числе произошедшими до вашего рождения.

Такой взгляд на мир озадачивал многих философов, задававшихся вопросом наличия у людей свободы воли. Однако в 1990-х годах начала развиваться интересная область физики, называемая *квантовой физикой* (или *квантовой механикой*), которая приводила в недоумение многих ученых. Оказывается, очень маленькие объекты (атомы и субатомные частицы) ведут себя совсем не так, как мы полагали, опираясь на законы так называемой классической физики. Судя по всему, в субатомном масштабе частицы иногда ведут себя как волны в том смысле, что

различные волны могут накладываться друг на друга, сливаясь в одну большую волну или отменяя друг друга на краткое мгновение.

Одним из измеряемых параметров таких частиц, как электроны, является *спин* (от spin — «вращение»). Например, мы можем выяснить, направлен спин электрона вверх или вниз. Пока ничего странного. Однако странности начинаются тогда, когда квантовая механика утверждает, что частица может находиться в двух этих состояниях *одновременно*, то есть иметь спин, направленный *и* вверх, *и* вниз. Тогда мы говорим, что частица находится в состоянии *квантовой суперпозиции*.

Это особое состояние можно вызвать вручную с помощью различных методов в зависимости от типа частицы. Она может оставаться в состоянии суперпозиции до тех пор, пока мы не выполним измерение, в результате которого она *коллапсирует* в одно из возможных состояний — «спин вверх» или «спин вниз». Именно квантовая суперпозиция лежит в основе принципа работы квантовых компьютеров. Вместо бита, который может принимать только одно значение — либо 1, либо 0, эти компьютеры используют *квантовый бит*, или *кубит*, значением которого может быть и 0, и 1 одновременно.

Еще более странно то, что, согласно квантовой теории, частица, находящаяся в суперпозиции, случайным образом выбирает итоговое состояние только в момент измерения, но не раньше, при этом вероятность наблюдения любого из состояний составляет 50 %. Если вас это удивляет, то вы не одиноки. Многие физики не могли себе представить, как это может работать в описанном ими детерминированном мире. Эйнштейн, убежденный в том, что с этой новой теорией что-то не так, однажды сказал: «Бог не играет в кости». Однако криптографы заинтересовались ею как способом получения *истинно* случайных чисел! Именно это и делают *квантовые генераторы случайных чисел* (КГСЧ), переводя такие частицы, как фотоны, в состояние суперпозиции, а затем измеряя их.

Физики также пытались понять, как квантовая механика могла бы работать в случае с объектами нашего масштаба. Результатом этих попыток стал знаменитый мысленный эксперимент с *котом Шрёдингера*, согласно которому кот в ящике одновременно является и мертвым, и живым до тех пор, пока наблюдатель не заглянет внутрь (что вызвало множество споров о том, что именно следует понимать под наблюдателем).

> В стальном сейфе находится кот и (защищенное от его непосредственного воздействия) устройство. Данное устройство состоит из счетчика Гейгера, внутри которого находится крошечное количество радиоактивного вещества. Это количество настолько мало, что за час может распасться один из атомов, но с такой же вероятностью может не распасться ни один из них. Если атом распадется, то счетчик через реле приведет в действие молоточек, который разобьет колбу с синильной кислотой. Если оставить всю эту систему на час, то можно сказать, что кот будет оставаться живым, если за это время не распадется ни один атом. Но первый же атомный распад приведет к отравлению кота. Пси-функция всей системы выразила бы это так, что живой и мертвый кот (если можно так сказать) являются смешанными или размазанными в равных пропорциях.
>
> *Эрвин Шрёдингер (The Present Situation in Quantum Mechanics, 1935)*

Все это кажется нам крайне контринтуитивным, потому что мы никогда не сталкивались с квантовым поведением в повседневной жизни. Теперь еще добавим странностей!

Иногда частицы взаимодействуют друг с другом (например, сталкиваются) и оказываются в состоянии такой сильной *корреляции*, при которой невозможно описать одну частицу без описания остальных. Это явление называется *квантовой запутанностью*, и оно является одним из секретных элементов, обеспечивающих повышенную производительность квантовых компьютеров. Если две частицы запутаны между собой, то при измерении состояние одной из них идеально коррелирует с состоянием другой. Чтобы лучше понять это, рассмотрим пример с двумя запутанными электронами. Если при измерении одного из них выяснится, что его спин направлен вверх, то мы точно сможем сказать, что спин другого направлен вниз, но до измерения первого электрона этот вывод сделать нельзя. Более того, результат любого подобного эксперимента всегда оказывается одинаковым.

Еще более поразительно то, что запутанность работает даже на очень больших расстояниях. Эйнштейн, Подольский и Розен утверждали, что описание квантовой механики неполно и в нем, скорее всего, отсутствуют *скрытые параметры*, которые могли бы объяснить запутанность (например, тем, что после разделения частицы точно знают, каким будет результат их измерения).

Эти ученые также описали мысленный эксперимент (по первым буквам их фамилий он назван *ЭПР-парадоксом*), в котором две запутанные частицы удаляются друг от друга на большое расстояние (исчисляемое в световых годах), а затем измеряются примерно в одно и то же время. Согласно принципам квантовой механики измерение одной из частиц оказывает мгновенное влияние на другую частицу, что невозможно, поскольку согласно теории относительности информация не может распространяться быстрее скорости света (отсюда и парадокс). Результат этого странного мысленного эксперимента Эйнштейн назвал «жутким действием на расстоянии».

Позднее Джон Белл вывел неравенство, ставшее известным как *теорема Белла*, которая в случае своей истинности доказала бы существование скрытых параметров, упомянутых авторами ЭПР-парадокса. Позднее это неравенство было многократно экспериментально опровергнуто, что убедило научное сообщество в реальности феномена квантовой запутанности и отсутствии каких-либо скрытых параметров.

Сегодня мы говорим, что измерение запутанных частиц приводит к их координации друг с другом, а это противоречит релятивистскому предсказанию о том, что информация не может передаваться быстрее скорости света. Действительно, если вы попробуете придумать способ использования запутанности для создания канала связи, то увидите, что это невозможно. Однако для криптографов «жуткое действие на расстоянии» обещает разработку новых способов обмена ключами. Эта идея называется *квантовым распределением ключей* (quantum key distribution, QKD).

Представьте, что вы передаете две запутанные частицы двум собеседникам, которые измеряют их, чтобы начать формировать один и тот же ключ, поскольку измерение одной частицы даст информацию о результатах измерения другой. Концепция QKD оказывается еще более привлекательной благодаря *теореме о запрете клонирования*, согласно которой вы не можете пассивно наблюдать за таким обменом и создать точную копию одной из частиц, передаваемых по этому каналу. Тем не менее подобные протоколы уязвимы для тривиальных атак посредника и, по большому счету, бесполезны при отсутствии способа аутентификации данных. Этот недостаток заставил некоторых криптографов, в том числе Брюса Шнайера, заявить: «QKD как продукт не имеет будущего».

На этом я завершу тему квантовой физики, поскольку для книги, посвященной криптографии, сказанного более чем достаточно. Если вы не верите ничему из только что прочитанного, вы не одиноки. В своей книге *Quantum Mechanics for Engineers* Леон ван Доммелен пишет: «Физика пришла к квантовой механике не потому, что она казалась наиболее логичным объяснением, а потому, что бесчисленные наблюдения сделали это неизбежным».

14.1.2. От создания квантовых компьютеров до квантового превосходства

Идея *квантовых вычислений* возникла в 1980 году. Пол Бениофф первым описал то, что может представлять собой квантовый компьютер, основанный на наблюдениях, сделанных за десятилетия развития квантовой механики. Позже в том же году Пол Бениофф и Ричард Фейнман заявили, что это единственный способ, позволяющий моделировать и анализировать квантовые системы, не сталкиваясь с ограничениями классических компьютеров.

Только 18 лет спустя компания IBM впервые продемонстрировала квантовый алгоритм, работающий на настоящем квантовом компьютере. В 2011 году компания D-Wave Systems, специализирующаяся на создании квантовых компьютеров, представила первый коммерчески доступный квантовый компьютер, дав старт развитию целой индустрии, направленной на создание первого масштабируемого квантового компьютера.

Однако для создания полезного квантового компьютера требуется сделать еще очень многое. На момент написания этой книги (2021 год) последним заметным результатом в этой сфере было сделанное в 2019 году компанией Google заявление о достижении *квантового превосходства* с помощью 53-кубитного квантового компьютера. Квантовое превосходство означает, что впервые в истории квантовый компьютер сделал нечто такое, чего не смог классический компьютер. За 3 мин 20 с он выполнил анализ, на который классическому компьютеру потребовалось бы около 10 000 лет. Чтобы вы не слишком радовались, я отмечу, что он превзошел классический компьютер в решении абсолютно бесполезной задачи. Тем не менее это невероятное достижение, и остается только гадать, к чему все это приведет.

Для выполнения вычислений квантовый компьютер использует явления квантовой физики, такие как суперпозиция и запутанность, точно так же, как классические компьютеры — электричество. Вместо битов в квантовых компьютерах задействуются *квантовые биты*, или *кубиты*, которые могут быть преобразованы с помощью *квантовых вентилей*, позволяющих задать им определенные значения или перевести в состояние суперпозиции и даже запутанности. Это чем-то похоже на применение вентилей в схемах классических компьютеров. После завершения вычисления кубиты можно измерить, чтобы интерпретировать их классическим способом — как нули и единицы. Затем, чтобы завершить полезное вычисление, результат можно интерпретировать с помощью классического компьютера.

В общем случае N запутанных кубитов содержат столько же информации, сколько $2N$ классических бит. Однако измерение кубитов в конце вычисления дает вам лишь N нулей или единиц. Таким образом, не всегда понятно, как квантовые компьютеры могут нам помочь, учитывая то, что они показали свою полезность лишь для ограниченного числа приложений. Возможно, они будут все более полезными по мере того, как люди станут находить нетривиальные способы применения их возможностей.

Уже сегодня вы можете использовать квантовый компьютер не выходя из дома. Такие сервисы, как IBM Quantum (https://quantum-computing.ibm.com), позволяют создавать квантовые схемы и запускать их на настоящих квантовых компьютерах, размещенных в облаке. Разумеется, пока (начало 2021 года) возможности таких сервисов весьма ограничены, учитывая то, что речь идет всего о нескольких кубитах. Тем не менее создание собственной схемы, которую можно совершенно бесплатно запустить на настоящем квантовом компьютере, — это потрясающий опыт.

14.1.3. Влияние алгоритмов Гровера и Шора на криптографию

К сожалению, как я уже говорил, квантовые компьютеры пригодны не для всех видов вычислений, а значит, они не являются более мощной альтернативой классическим компьютерам. Но тогда для чего же они нужны?

В 1994 году, когда концепция квантового компьютера была всего лишь мысленным экспериментом, Питер Шор предложил квантовый алгоритм для решения задач дискретного логарифмирования и факторизации. Ему пришло в голову, что квантовый компьютер можно использовать для быстрого выполнения вычислений, связанных с решением трудных задач, встречающихся в криптографии. Оказалось, что существует эффективный квантовый алгоритм, который позволяет найти для любого заданного x такой *период*, что $f(x + период) = f(x)$. Например, найти значение *периода* такое, что $g^{x + период} = g^x \bmod N$. Это, в свою очередь, приводит к алгоритмам, позволяющим эффективно решать задачи факторизации и дискретного логарифмирования, что напрямую затрагивает такие алгоритмы, как RSA (см. главу 6) и алгоритм Диффи — Хеллмана (см. главу 5).

Алгоритм Шора губителен для *асимметричной криптографии*, поскольку большинство используемых сегодня асимметричных алгоритмов опираются на задачу дискретного логарифмирования или факторизации, а это почти все, о чем говорилось в этой книге. Можно было бы подумать, что дискретное логарифмирование и факторизация по-прежнему остаются трудными математическими задачами и что мы могли бы увеличить параметры наших алгоритмов, чтобы защититься от квантовых компьютеров. Но, к сожалению, в 2017 году Бернштейн и другие исследователи показали, что увеличение параметров хоть и работает, но крайне непрактично. Исследователи подсчитали, что алгоритм RSA можно сделать квантово-устойчивым, увеличив его параметры до 1 Тбайт, что, мягко говоря, нереально.

> Алгоритм Шора разрушает криптографические системы с открытым ключом, в основе которых лежат алгоритм RSA и задачи дискретного логарифмирования в группе конечного поля и на эллиптической кривой. Таким конфиденциальным документам, как медицинские карты пациентов и государственные секреты, должна быть гарантирована долгосрочная безопасность, однако информация, шифруемая сегодня с помощью алгоритма RSA или эллиптических кривых, после появления квантовых компьютеров будет расшифрована так же легко, как сегодня расшифровываются сообщения, некогда зашифрованные с помощью «Энигмы».
>
> *PQCRYPTO: Initial recommendations of long-term secure post-quantum systems (2015)*

В сфере *симметричной криптографии* ситуация не столь тревожная. Алгоритм Гровера был предложен в 1996 году Ловом Гровером в качестве способа оптимизации поиска в неупорядоченном списке. В среднем поиск в неупорядоченном списке из N элементов требует выполнения $N/2$ операций на классическом компьютере, а на квантовом компьютере для этого потребовалось бы всего $\sqrt{N}$ операций. Весьма впечатляющее ускорение!

Алгоритм Гровера — это довольно универсальный инструмент, который может применяться в криптографии самыми разными способами, например, для извлечения симметричного ключа шифрования или поиска коллизий хеш-функции. Для нахождения ключа длиной 128 бит с помощью алгоритма Гровера на квантовом компьютере потребовалось бы выполнить 2^{64} операций, в то время как на классическом компьютере для этого потребовалось бы 2^{127} операций. Это звучит довольно зловеще для всех алгоритмов симметричной криптографии, однако для того, чтобы защититься от подобной атаки, достаточно просто увеличить параметры безопасности с 128 до 256 бит. Таким образом, если вы хотите защитить свою симметричную криптографическую систему от квантовых компьютеров, можете просто использовать SHA-3-512 вместо SHA-3-256, AES-256GCM вместо AES-128-GCM и т. д.

Подводя итог, можно сказать, что с симметричной криптографией все более или менее в порядке, а с асимметричной — не очень. На самом деле ситуация даже хуже, чем может показаться на первый взгляд, поскольку использованию алгоритмов симметричной криптографии часто предшествует обмен ключами, уязвимый для квантовых компьютеров. Неужели привычной нам криптографии пришел конец?

14.1.4. Постквантовая криптография, защита от квантовых компьютеров

К счастью, для криптографии конец света еще не настал. Сообщество быстро отреагировало на квантовую угрозу и приступило к исследованию старых и новых алгоритмов, *неуязвимых* для атак Шора и Гровера. Так родилась область *квантово-устойчивой криптографии*, называемая также *постквантовой криптографией*. Работу по стандартизации ведут различные организации, наиболее многообещающими являются усилия института NIST, который в 2016 году приступил к разработке стандартов постквантовой криптографии.

> Судя по всему, переход к постквантовой криптографии не будет легким, поскольку у нас нет готовых альтернатив нынешним криптографическим алгоритмам с открытым ключом. Нам потребуется потратить значительные усилия на разработку, стандартизацию и развертывание новых постквантовых криптосистем. Кроме того, этот переход должен произойти задолго до создания крупномасштабных квантовых компьютеров, чтобы любая информация, которая впоследствии будет скомпрометирована с помощью квантового криптоанализа, перестала быть чувствительной еще до момента ее компрометации. Таким образом, планировать этот переход желательно заранее.
>
> *Проект стандартов постквантовой криптографии NIST (2016)*

С тех пор как организация NIST начала этот процесс, заявки подали 82 кандидата и было проведено три раунда, в результате чего список кандидатов сократился до семи финалистов и восьми альтернативных финалистов (они вряд ли будут стандартизированы, но достаточно уникальны для того, чтобы стать хорошим вариантом, если одна из парадигм, используемых финалистами, перестанет быть рабочей). Усилия NIST по стандартизации направлены на замену наиболее распространенных типов асимметричных криптографических примитивов, к которым относятся схемы цифровой подписи и асимметричного шифрования. Последний также может служить примитивом, используемым для обмена ключами, как было сказано в главе 6.

В оставшейся части этой главы я расскажу о различных типах постквантовых криптографических алгоритмов, которые рассматриваются в качестве вариантов для стандартизации, и выделю те, которые вы можете использовать уже сегодня.

14.2. ПОДПИСИ НА ОСНОВЕ ХЕША: НЕ НУЖНО НИЧЕГО, КРОМЕ ХЕШ-ФУНКЦИИ

Хотя все применяемые на практике схемы цифровой подписи, по-видимому, используют хеш-функции, существуют способы построения схем, в которых задействуются только хеш-функции и ничего больше. Что еще лучше, эти схемы, как правило, полагаются только на устойчивость хеш-функций к нахождению прообраза, а не на устойчивость к коллизиям. Это довольно привлекательная идея, учитывая то, что в основе огромной части прикладной криптографии лежат надежные и хорошо изученные хеш-функции.

Современные хеш-функции неуязвимы и для квантовых компьютеров, что делает квантово-устойчивыми схемы цифровой подписи на их основе. Давайте посмотрим, что представляют собой подписи на основе хеша и как они работают.

14.2.1. Одноразовые подписи с подписями Лэмпорта

Восемнадцатого октября 1979 года Лесли Лэмпорт опубликовал свою концепцию *одноразовых подписей* (one-time signature, OTS), представляющих собой пары ключей, которые можно использовать для подписи только один раз. Безопасность большинства схем цифровой подписи частично обеспечивается односторонними функциями (обычно хеш-функциями). Прелесть схемы Лэмпорта заключается в том, что ее подпись опирается исключительно на безопасность таких односторонних функций.

Представьте, что вы хотите подписать 1 бит. Сначала вы генерируете пару ключей следующим образом.

1. Генерируете два случайных числа, x и y, которые будут выступать в качестве закрытого ключа.
2. Хешируете x и y для получения двух дайджестов, $h(x)$ и $h(y)$, которые можно опубликовать в качестве открытого ключа.

Чтобы подписать бит, установленный в 0, раскройте часть x своего закрытого ключа; чтобы подписать бит, установленный в 1, раскройте часть y. Для проверки подписи выполните хеширование, чтобы убедиться в ее соответствии правильной части открытого ключа. Я проиллюстрировал этот процесс на рис. 14.1.

Если подписание бита не кажется вам особенно полезным, ничего страшного. Подпись Лэмпорта можно применить и к более объемным входным данным путем создания большего количества пар секретов, по одному на бит (рис. 14.2). Очевидно, что если размер входных данных превышает 256 бит, то сначала их нужно хешировать, а затем подписывать.

Основное ограничение этой схемы заключается в том, что можно применять ее только для однократного подписания — при двукратном подписании вы предоставляете третьим лицам право смешивать две подписи, чтобы подделать другие действительные подписи. Мы можем улучшить эту ситуацию наивным способом, генерируя большое количество одноразовых пар ключей вместо одной и удаляя каждую из них сразу после применения. Однако это не только делает ваш открытый ключ таким же большим, как количество подписей, которые вы намерены использовать, но и означает, что нужно отслеживать, какие из пар ключей задействовали, или, что еще лучше, избавляться от использованных закрытых ключей. Например, если вы намерены подписать максимум 1000 сообщений длиной 256 бит с помощью хеш-функции, генерирующей выходное значение размером 256 бит, то размер закрытого и открытого ключа должен составлять $1000 \times (256 \times 2 \times 256)$ бит, то есть около 16 Мбайт. Это довольно много для 1000 подписей.

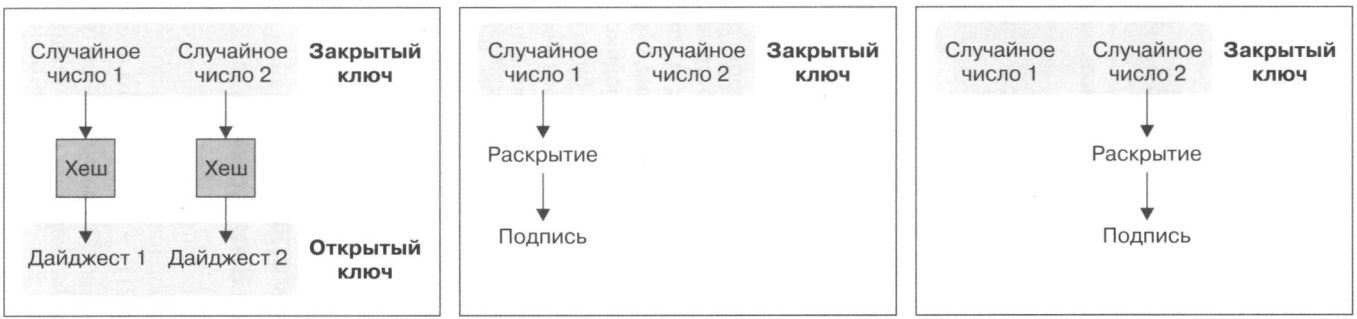


Рис. 14.1. Подпись Лэмпорта — это одноразовая подпись (OTS), основанная только на хеш-функциях. Чтобы сгенерировать пару ключей для подписания бита, сгенерируйте два случайных числа, которые станут вашим закрытым ключом, и хешируйте каждое из них по отдельности, чтобы получить два дайджеста своего открытого ключа. Чтобы подписать бит, установленный в 0, раскройте первое случайное число; чтобы подписать бит, установленный в 1, раскройте второе случайное число

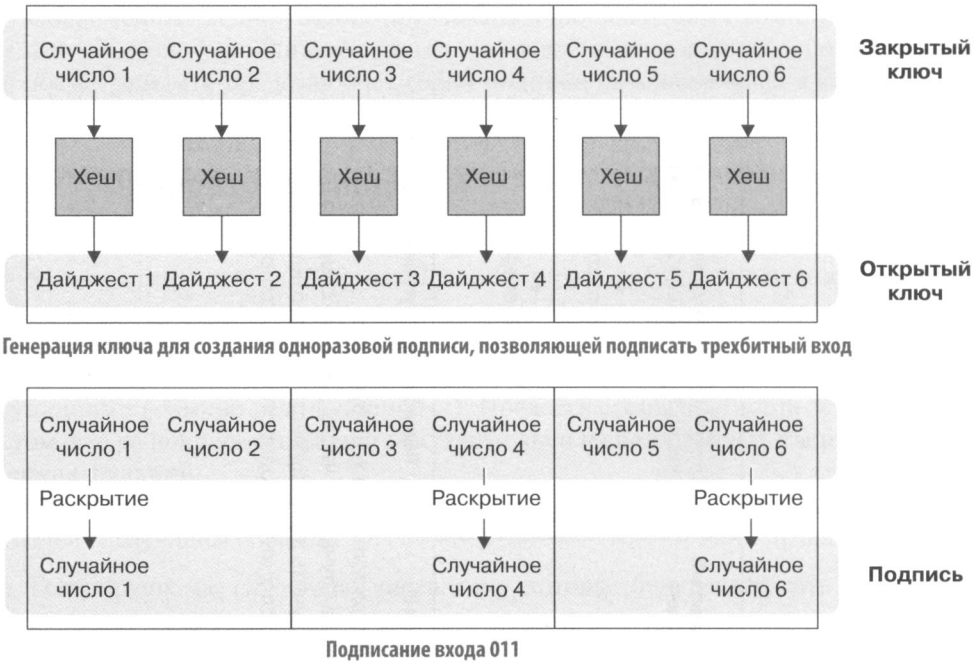


Рис. 14.2. Чтобы сгенерировать пару ключей подписи Лэмпорта для подписания n-битного сообщения, сгенерируйте 2n случайных чисел, которые станут вашим закрытым ключом, и хешируйте каждое из них по отдельности, чтобы получить 2n дайджестов открытого ключа. Чтобы подписать сообщение, переберите все пары секретов и n бит, раскрывая первый элемент для подписания бита, установленного в 0, или второй элемент — для подписания бита, установленного в 1

Большинство предлагаемых сегодня схем цифровой подписи на базе хеш-функций опираются на основы, заложенные Лэмпортом, и позволяют создавать гораздо большее количество подписей (иногда практически неограниченное), предусматривают закрытые ключи без сохранения состояния (хотя в некоторых схемах все еще фигурируют ключи с сохранением состояния) и более практичные размеры параметров.

14.2.2. Ключи меньшего размера с одноразовыми подписями Винтерница

Через несколько месяцев после публикации работы Лэмпорта Роберт Винтерниц с математического факультета Стэнфордского университета предложил публиковать хеши хешей секретов $h(h(\cdots h(x))) = h^w(x)$ вместо нескольких дайджестов нескольких секретов, чтобы оптимизировать размер закрытого ключа. Эта схема была названа *одноразовой подписью Винтерница* (Winternitz one-time signature, WOTS) в честь ее автора.

Например, выбрав $w = 16$, вы можете подписать 16 различных значений или входов размером по 4 бита. Сначала генерируете случайное значение x, которое служит вашим закрытым ключом, и хешируете его 16 раз для получения открытого ключа $h^{16}(x)$. Теперь представьте, что вы хотите подписать бит 1001 (что соответствует числу 9 в десятичной системе счисления). Для этого публикуете результат девятой итерации хеширования $h^9(x)$. Этот процесс проиллюстрирован на рис. 14.3.

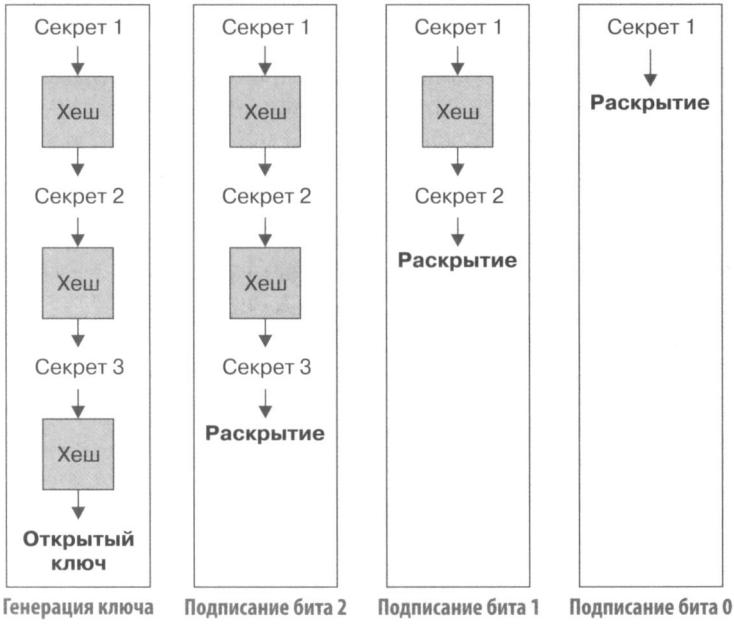


Рис. 14.3. Схема одноразовой подписи Винтерница оптимизирует схему подписи Лэмпорта за счет использования только одного секрета, который итеративно хешируется для получения множества других секретов и, наконец, открытого ключа. Раскрытие другого секрета позволяет подписать другое число

Потратьте несколько минут на то, чтобы понять, как работает эта схема. Заметили ли вы в ней какие-то проблемы? Одна из главных проблем заключается в том, что эта схема допускает *подделку подписи*. Представьте, что вы видите чужую подпись для бита 1001, которая в соответствии с предыдущим примером представляет собой результат итерации $h^9(x)$. Вы можете просто хешировать ее, чтобы получить результат любых других итераций, например, $h^{10}(x)$ или $h^{11}(x)$, а значит, действительную подпись для бита 1010 или 1011. Эту проблему можно обойти, добавив после сообщения короткий тег аутентификации, который вам также нужно будет подписать (рис. 14.4). Чтобы убедиться в том, что это решает проблему подделки подписи, попробуйте подделать подпись на основе другой подписи.

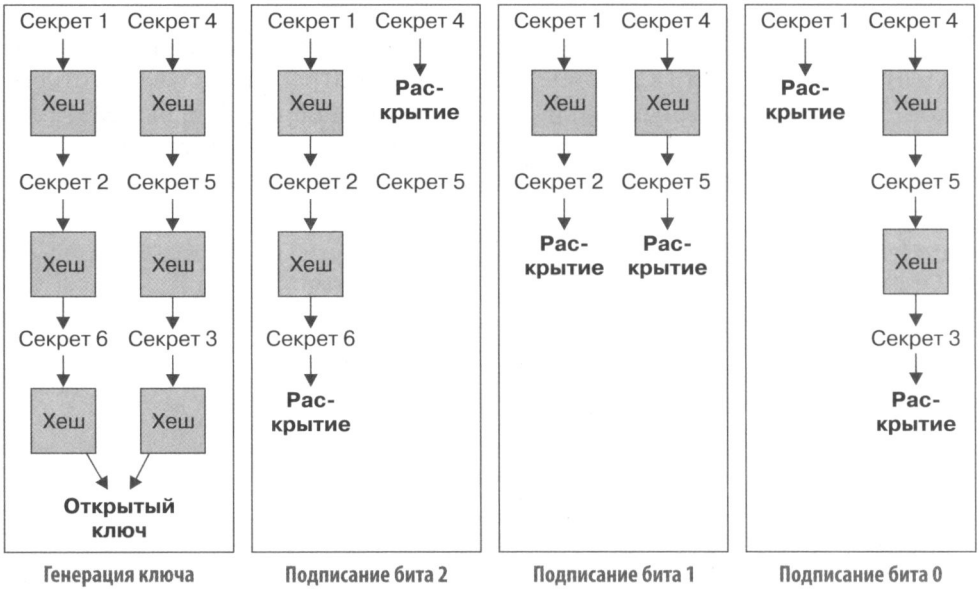


Рис. 14.4. В схеме WOTS используется дополнительный ключ для проверки подлинности подписи с целью предотвращения ее фальсификации. Это работает следующим образом: первым закрытым ключом подписывается сообщение, а вторым — дополнение к нему. Вам уже должно быть очевидно, что в любом из описанных сценариев подделка подписи не может привести к созданию новой действительной подписи

14.2.3. Многоразовые подписи с XMSS и SPHINCS+

До сих пор мы рассматривали способы подписания, основанные на одних только хеш-функциях. Хотя схема подписи Лэмпорта работает, проблема заключается в слишком большом размере ключей, но ее в некоторой степени решает схема WOTS. Тем не менее обе эти схемы плохо масштабируются, поскольку предполагают создание одноразовых подписей (повторное использование пары ключей приводит к нарушению схемы), поэтому их параметры линейно зависят от количества подписей, которые вы намерены применить.

Схемы, допускающие повторное использование пары ключей для создания не одной, а нескольких подписей, существуют. Они называются схемами *многоразовой подписи* (few-time signature, FTS), однако тоже портятся, допуская подделку подписи, если задействуются слишком много раз. Схемы FTS полагаются на низкую вероятность повторного применения одной и той же комбинации секретов, выбранной из их пула. Это уже некоторое улучшение по сравнению с одноразовыми подписями, позволяющими снизить риск повторного использования ключей. Но мы можем добиться большего.

Какая техника из описанных в этой книге позволяет сжать множество вещей в одну? Ответ: деревья Меркла. Как было сказано в главе 12, *дерево Меркла* — это

структура данных, которая дает короткий ответ на вопрос: «Присутствуют ли мои данные в этом множестве?» В 1990-х годах Меркл, разработавший эту структуру данных, изобрел также схему подписи на основе хеш-функций, которая сжимает несколько одноразовых подписей в дерево Меркла.

Суть ее довольно проста: каждый лист дерева содержит хеш одноразовой подписи, а корневой хеш можно использовать в качестве открытого ключа, сократив его размер до размера выходного значения вашей хеш-функции. Чтобы что-то подписать, выбираете одноразовую подпись, которую не применяли ранее, а затем применяете ее так, как показано в подразделе 14.2.2. Подпись представляет собой одноразовую подпись, сопровождаемую доказательством того, что она принадлежит вашему дереву Меркла (для доказательства задействуются все соседние узлы). Эта схема, очевидно, предполагает сохранение состояния, так как вам нужно следить за тем, чтобы ни одна из одноразовых подписей в дереве не применялась повторно (рис. 14.5).

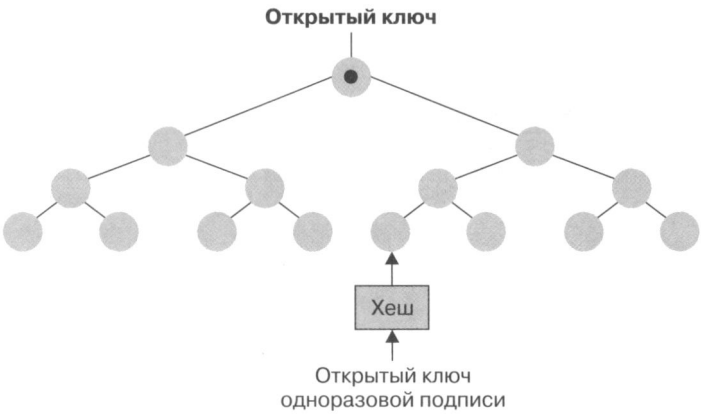


Рис. 14.5. Подпись Меркла — это основанный на хешировании алгоритм с сохранением состояния, который использует дерево Меркла для сжатия множества открытых ключей одноразовой подписи (OTS) в открытый ключ меньшего размера (корневой хеш). Чем больше дерево, тем больше подписей оно позволяет создать. Обратите внимание на то, что теперь подписи имеют накладные расходы, связанные с доказательством принадлежности, которое представляет собой количество соседних узлов, позволяющих убедиться в том, что связанная с подписью OTS является частью дерева

Расширенная схема подписи Меркла (extended Merkle signature scheme, XMSS), стандартизированная в документе RFC 8391, добавила ряд оптимизаций к исходной схеме. Например, чтобы создать пару ключей, позволяющую подписать N сообщений, необходимо сгенерировать N закрытых ключей OTS. Хотя открытый ключ теперь представляет собой просто корневой хеш, все равно приходится хранить N закрытых ключей OTS. Схема XMSS позволяет уменьшить размер закрытого ключа путем детерминированной генерации каждой OTS в дереве с использованием зерна и позиции листа. Благодаря этому вместо хранения всех закрытых ключей OTS

вы можете хранить только зерно в качестве закрытого ключа и быстро восстанавливать любую пару ключей OTS на основе ее положения в дереве и зерна. Для отслеживания того, какой из листов/OTS был использован последним, закрытый ключ предусматривает счетчик, значение которого инкрементируется каждый раз, когда с его помощью создается подпись.

Однако в дереве Меркла можно хранить ограниченное количество OTS. Чем больше дерево, тем больше времени потребуется на его восстановление для подписания сообщений, поскольку для получения доказательства Меркла необходимо восстановить все листья. Чем меньше дерево, тем меньше закрытых ключей OTS требуется восстанавливать при подписании, но это, очевидно, противоречит нашей цели, так как мы снова сталкиваемся с ограниченным количеством подписей. Решение заключается в использовании меньшего дерева, OTS в листьях которого применяются для подписания не сообщений, а корневых хешей других деревьев Меркла с OTS. Это превращает исходное дерево в *гипердерево* (дерево деревьев) и является одним из вариантов схемы XMSS под названием XMSS^{MT}. При использовании XMSS^{MT} необходимо восстановить только деревья, задействованные в пути, ведущем к нужной OTS (рис. 14.6).

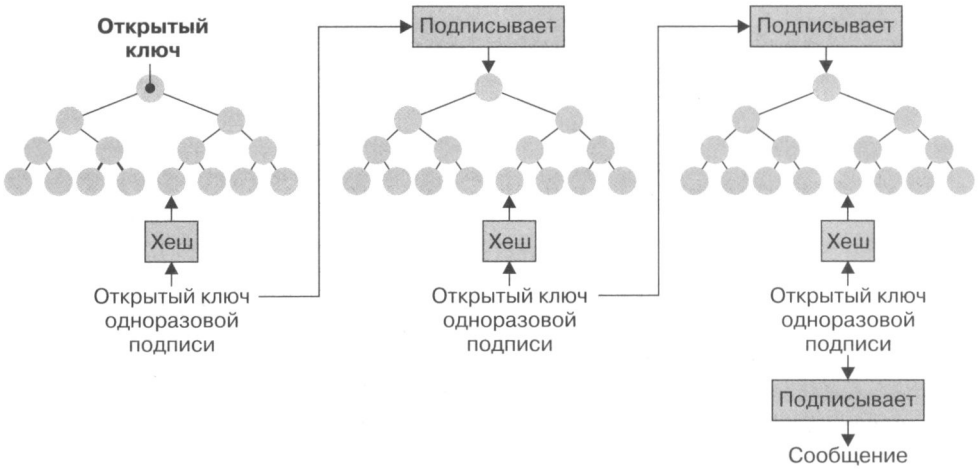


Рис. 14.6. Схема цифровой подписи XMSS^{MT} на основе хеш-функции с сохранением состояния подразумевает несколько деревьев для увеличения количества подписей, поддерживаемых схемой, при одновременном сокращении трудозатрат на генерацию ключей и времени подписания сообщений. Каждое дерево детерминированным образом генерируется только в том случае, если задействуется на пути к последнему листу, содержащему ключ OTS для подписания сообщения

Обратите внимание на то, что в некоторых ситуациях сохранение состояния, присущее схемам XMSS и XMSS^{MT}, может не вызывать проблем, но в целом это нежелательное свойство. Необходимость отслеживать показание счетчика контринтуитивна, поскольку от тех, кто пользуется распространенными схемами цифровой подписи, это обычно не требуется. Такое изменение практики может привести к повторному применению OTS и, следовательно, к подделке подписи.

Например, откат к предыдущему состоянию файловой системы или использование одного и того же ключа подписи на нескольких серверах может привести к тому, что один и тот же путь в гипердереве будет дважды задействован для подписания сообщения.

Чтобы исправить один из самых больших недостатков схемы XMSS (сохранение состояния) и предоставить интерфейс, похожий на привычные нам схемы подписи, в рамках конкурса алгоритмов постквантовой криптографии, проводимого NIST, была предложена *схема подписи SPHINCS+*. Эта схема цифровой подписи без сохранения состояния привнесла в XMSS^{MT} три основных изменения.

- *Результатом двукратного подписания одного и того же сообщения является одна и та же подпись.* Как и в случае с EdDSA (см. главу 7), путь, используемый в гипердереве, детерминируется закрытым ключом и сообщением. Это гарантирует, что двукратное подписание одного и того же сообщения приводит к одной и той же OTS и, следовательно, к созданию одной и той же подписи. А если вы подписываете сообщения других людей, то благодаря применению закрытого ключа злоумышленники не смогут предсказать, какой путь вы выберете для подписания этих сообщений.
- *Использование большего количества деревьев.* Схема XMSS^{MT} позволяет избежать повторного применения одной и той же OTS благодаря отслеживанию того, какая из OTS использована последней. Поскольку схема SPHINCS+ была разработана в первую очередь для избавления от необходимости отслеживать состояние, она должна избегать коллизий при псевдослучайном выборе пути. Для этого схема просто использует гораздо большее количество OTS, тем самым уменьшая вероятность их повторного применения. А поскольку схема SPHINCS+ также использует гипердерево, это ведет к увеличению количества деревьев.
- *Использование многоразовых подписей* (FTS). Поскольку безопасность схемы основана на вероятности повторного задействования одного и того же пути, SPHINCS+ предполагает также замену конечной OTS, используемой для подписания сообщений, на FTS, о которой я говорил ранее. Таким образом, повторное применение одного и того же пути для подписания двух разных сообщений все равно не приводит к нарушению схемы цифровой подписи.

Хотя схема SPHINCS+ рассматривается в качестве кандидата на стандартизацию в рамках конкурса NIST по постквантовой криптографии, это не главный претендент. Она работает очень медленно и создает слишком большие подписи по сравнению с предлагаемыми альтернативами (в частности, по сравнению со схемами на основе решеток, о которых вы узнаете далее в этой главе). Схемы цифровой подписи на основе хеш-функций с сохранением состояния, такие как XMSS, обеспечивают более высокую скорость и более компактные размеры подписей — менее 3 Кбайт по сравнению с минимальным размером 8 Кбайт у SPHINCS+. (Что касается размеров открытых ключей, то обе эти схемы сопоставимы с такими доквантовыми схемами цифровой подписи, как ECDSA и Ed25519.) Благодаря более реалистичным размерам параметров и хорошо проработанной системе безопасности схема XMSS была рекомендована организацией NIST в качестве

предварительного стандарта в документе SP 800-208. Recommendation for Stateful Hash-Based Signature Schemes.

Далее рассмотрим два других способа построения квантово-устойчивых криптографических примитивов. Однако должен вас предупредить: они предполагают использование гораздо более изощренной математики!

14.3. СОКРАЩЕНИЕ ДЛИНЫ КЛЮЧЕЙ И ПОДПИСЕЙ С ПОМОЩЬЮ КРИПТОГРАФИИ НА РЕШЕТКАХ

Большое количество схем постквантовой криптографии основано на решетках — математических структурах, о которых вы узнаете в этом разделе. К схемам данного типа принадлежит половина финалистов конкурса постквантовой криптографии, проводимого организацией NIST, что делает криптографию на решетках наиболее вероятным кандидатом на стандартизацию. В этом разделе я расскажу о двух алгоритмах на основе решеток: схеме цифровой подписи Dilithium и примитиве шифрования с открытым ключом Kyber. Однако перед этим нужно разобраться с самим понятием решетки.

14.3.1. Что такое решетка

Во-первых, термин «криптография на решетках» означает совсем не то, что вы думаете. Возьмем, к примеру, алгоритм RSA (см. главу 6). Мы говорим, что он основан на задачах факторизации. Однако это не означает, что в RSA используется факторизация, — это означает, что факторизация является способом атаки на этот алгоритм, а поскольку задача факторизации сложна, мы считаем его безопасным. То же самое можно сказать о криптосистемах на основе решеток. *Решетки* — это структуры, которые предусматривают труднорешаемые задачи, и пока их решение остается трудным, основанные на них криптосистемы остаются безопасными.

Итак, что же такое решетка? Она напоминает *векторное пространство*, только с целыми числами. Напомню, что векторное пространство представляет собой множество всех векторов, которые могут быть созданы с использованием:

- *базиса* — набора векторов, например: (0, 1) и (1, 0);
- *операции с векторами*. Векторы можно складывать, например: (0, 1) + (1, 0) = (1, 1);
- *скалярной операции*. Вектор можно умножить на так называемый скаляр, например: 3 × (1, 2) = (3, 6).

В нашем примере векторное пространство содержит все векторы, которые могут быть выражены в виде линейной комбинации базисных векторов. Это относится к любому вектору, который может быть записан в виде $a(0, 1) + b(1, 0)$ для любых скаляров a и b. Например, 0,5 × (0, 1) + 3,87 × (1, 0) = (3,87, 0,5) находится в нашем векторном пространстве, как и 99 × (0, 1) + 0 × (1, 0) = (0, 99), и т. д.

Решетка — это векторное пространство, в котором все числа целые. Да, мы любим использовать целые числа в криптографии. Я проиллюстрировал это на рис. 14.7.

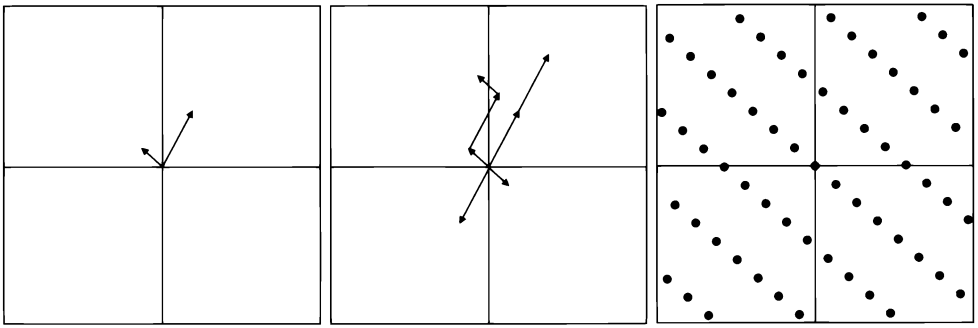


Рис. 14.7. *Слева* изображен базис, состоящий из двух векторов. Решетка может быть сформирована взятием всех возможных целочисленных линейных комбинаций этих двух векторов *(в середине)*. Полученную решетку можно интерпретировать как узор из точек, вечно повторяющийся в пространстве *(справа)*

В теории решеток существует несколько хорошо известных трудных задач, для решения которых есть специальный алгоритм. Часто такие алгоритмы оказываются лучшим из того, что нам удалось придумать, но это не обязательно делает их эффективными или даже практичными. Таким образом, эти задачи считаются трудными по крайней мере до тех пор, пока для них не будет найдено более эффективное решение. Вот две самые трудные задачи (рис. 14.8).

- *Задача поиска кратчайшего вектора* (shortest vector problem, SVP) сводится к нахождению самого короткого ненулевого вектора в решетке.
- *Задача поиска ближайшего вектора* (closest vector problem, CVP) сводится к тому, чтобы при заданной координате, не лежащей на решетке, найти ближайшую к этой координате точку на решетке.

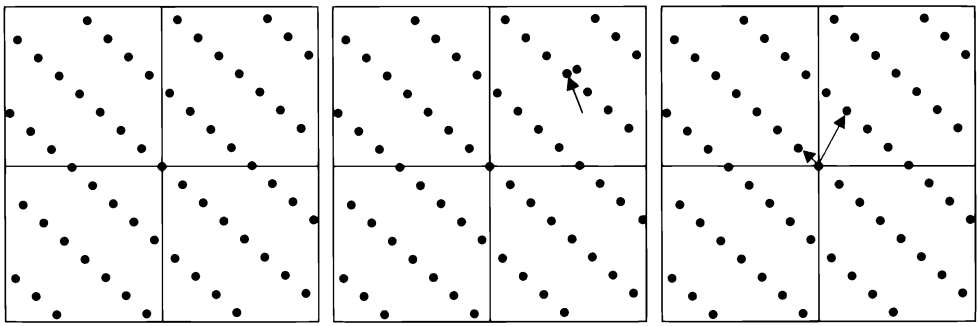


Рис. 14.8. Иллюстрация двух основных задач теории решеток, используемых в криптографии: задачи поиска ближайшего вектора и задачи поиска кратчайшего вектора

Обычно для решения обеих задач используются алгоритмы LLL (алгоритм Ленстры — Ленстры — Ловаса) или BKZ (блочный алгоритм Коркина — Золотарева), так как задача CVP может быть сведена к SVP. Эти алгоритмы сокращают базис решетки, то есть пытаются найти набор таких векторов, которые были бы короче заданных и смогли бы создать точно такую же решетку.

14.3.2. Обучение с ошибками — основа для криптографических систем?

В 2005 году Одед Регев представил концепцию *обучения с ошибками* (learning with errors, LWE), которая легла в основу многих криптографических схем, включая некоторые алгоритмы, описанные в этой главе. Прежде чем продолжить, давайте разберемся, в чем заключается суть концепции LWE. Начнем со следующих уравнений, которые представляют собой линейные комбинации одних и тех же целых чисел s_0 и s_1:

$$5s_0 + 2s_1 = 27;$$

$$2s_0 + 0s_1 = 6.$$

Мы знаем, что при наличии достаточного количества этих уравнений можно быстро и эффективно узнать значения s_0 и s_1 с помощью алгоритма *исключения Гаусса*. Однако, если привнести в эти уравнения немного шума, задача станет гораздо сложнее:

$$5s_0 + 2s_1 = 28;$$

$$2s_0 + 0s_1 = 5.$$

Хотя ответ нетрудно найти и для более шумных уравнений, при возрастании используемых чисел и количества s_i эта задача становится очень сложной.

В этом и заключается суть задачи LWE, только она, как правило, описывается с помощью векторов. Представьте, что у вас есть секретный вектор **s** с координатами по модулю некоторого большого числа. Можете ли вы найти значение **s** при наличии произвольного количества случайных векторов $\mathbf{a}_i$ одинакового размера в соответствии с выражением $\mathbf{a}_i s + e_i$, где e_i — случайная малая ошибка?

ПРИМЕЧАНИЕ

Произведение двух векторов, **v** и **w**, можно вычислить с помощью *скалярного произведения*, которое представляет собой сумму произведений каждой пары координат. Например, если $\mathbf{v} = (v_0, v_1)$ и $\mathbf{w} = (w_0, w_1)$, то $\mathbf{vw} = v_0w_0 + v_1w_1$.

Так, если я использую секрет **s** = (3, 6) и передам вам случайные векторы $\mathbf{a}_0$ = (5, 2) и $\mathbf{a}_1$ = (2, 0), то получу уравнения, приведенные в начале раздела. Как я уже говорил, схемы, основанные на решетках, на самом деле не используют сами решетки. Решетки применяются скорее для подтверждения безопасности этих схем, пока задача SVP остается трудно решаемой. Это можно увидеть, только если записать предыдущие уравнения в матричной форме (рис. 14.9).

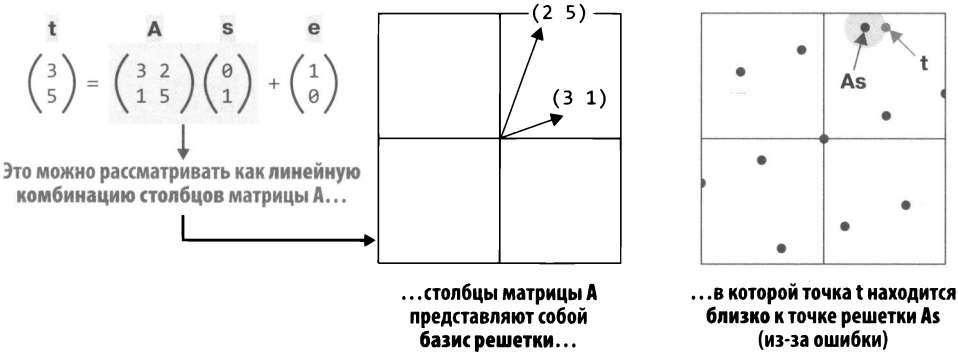


Рис. 14.9. Задача обучения с ошибками считается решетчатой конструкцией из-за возможности ее сведения к задаче поиска ближайшего вектора (CVP). Другими словами, если мы можем решить задачу CVP, то можем найти решение и для задачи LWE

Матричная форма очень важна, поскольку она значительно упрощает объяснение большинства LWE-схем. Потратьте несколько минут на освоение темы матричного умножения. Кроме того, если вы не заметили, я использовал условные обозначения, которые очень помогают при чтении уравнений с матрицами и векторами: и те и другие выделены жирным шрифтом, а матрицы всегда обозначаются прописными буквами. Например, $\mathbf{A}$ — это матрица, $\mathbf{a}$ — вектор, а b — просто число.

ПРИМЕЧАНИЕ

Существует несколько вариантов задачи LWE, например обучение с ошибками в кольце (ring-LWE) и обучение с ошибками с округлением по модулю (module-LWE), которые, по сути, являются одной и той же задачей, но с координатами, принадлежащими группам различного типа. Эти варианты часто предпочтительны благодаря компактности и оптимизации, которые они обеспечивают. Разница между вариантами задачи LWE никак не влияет на приведенные далее объяснения.

Теперь, когда вы знаете, что такое задача LWE, давайте познакомимся с основанной на ней постквантовой криптографической системой *CRYSTALS* (Cryptographic Suite for Algebraic Lattices — криптографический набор [алгоритмов] для алгебраических решеток). Набор CRYSTALS включает в себя два криптографических примитива: обмен ключами *Kyber* и схему цифровой подписи *Dilithium*.

14.3.3. Kyber — обмен ключами на основе решеток

Две схемы, дошедшие до финала конкурса NIST, тесно связаны между собой: CRYSTALS-Kyber и CRYSTALS-Dilithium были предложены одной и той же командой исследователей и обе основаны на задаче LWE. *Kyber* — это примитив шифрования с открытым ключом, который может использоваться для обмена ключами, как будет показано в этом разделе. *Dilithium* — это схема цифровой подписи,

о которой мы поговорим в следующем разделе. Также учтите, что, поскольку эти алгоритмы все еще находятся в процессе разработки, я изложу только основные концепции, которые лежат в основе обеих схем.

Предположим, что все операции выполняются в группе целых чисел по модулю большого числа q. Предположим также, что ошибки и закрытые ключи *сэмплируются* (выбираются равномерно случайным образом) из небольшого диапазона с центром в точке 0, который мы будем называть *диапазоном ошибок*. В частности, диапазон ошибок представляет собой диапазон $[-B, B]$, где B намного меньше q. Это важно, поскольку для того, чтобы считаться ошибками, некоторые члены должны быть меньше некоторого значения.

Чтобы сгенерировать закрытый ключ, просто сгенерируйте случайный вектор $\mathbf{s}$, каждый коэффициент которого находится в диапазоне ошибок. Первая часть открытого ключа представляет собой список случайных векторов $\mathbf{a}_i$ одинакового размера, а вторая — связанный с ним список зашумленных скалярных произведений $\mathbf{t}_i = \mathbf{a}_i s + e_i \bmod q$. Это и есть задача LWE, о которой мы говорили ранее. Для упрощения дальнейшего объяснения можем представить ее в матричной форме:

$$\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e},$$

где матрица $\mathbf{A}$ содержит случайные векторы $\mathbf{a}_i$ в качестве строк, а вектор ошибок $\mathbf{e}$ содержит отдельные значения ошибок $\mathbf{e}_i$.

Чтобы выполнить обмен ключами по схеме Kyber, зашифровываем симметричный ключ длиной 1 бит (да, 1 бит!). Это напоминает механизм инкапсуляции ключей RSA, который рассматривался в главе 6. Процесс шифрования состоит из следующих четырех шагов.

1. Генерируем вектор эфемерного закрытого ключа $\mathbf{r}$ (коэффициенты которого находятся в диапазоне ошибок) и связанный с ним эфемерный открытый ключ $\mathbf{r}\mathbf{A} + \mathbf{e}_1$ с некоторым случайным вектором ошибок $\mathbf{e}_1$, используя матрицу $\mathbf{A}$ другого собеседника в качестве открытого параметра. Обратите внимание на то, что умножение матрицы выполняется справа, что предполагает умножение вектора $\mathbf{r}$ на столбцы $\mathbf{A}$ вместо вычисления $\mathbf{A}\mathbf{r}$ (умножения вектора $\mathbf{r}$ на строки $\mathbf{A}$). Это мелочь, но она необходима для правильной расшифровки.
2. Сдвигаем наше сообщение влево, умножая его на $q/2$, чтобы исключить влияние небольших ошибок на это сообщение. Обратите внимание на то, что $q/2$ по модулю q обычно означает q, умноженное на величину, обратную 2 по модулю q, но здесь это просто означает ближайшее целое число к $q/2$.
3. Вычисляем общий секрет с помощью скалярного произведения нашего эфемерного закрытого ключа и открытого ключа собеседника.
4. Зашифровываем (сдвинутое) сообщение, складывая его с общим секретом и случайной ошибкой $\mathbf{e}_2$. В результате получаем шифротекст.

После выполнения этих действий мы можем отправить эфемерный открытый ключ и шифротекст своему собеседнику. А после получения эфемерного открытого

ключа и шифротекста можем выполнить следующие шаги для расшифровки сообщения.

1. Получить общий секрет, вычислив скалярное произведение своего секрета и полученного эфемерного открытого ключа.
2. Вычесть общий секрет из шифротекста (результат будет состоять из сдвинутого сообщения и некоторой ошибки).
3. Вернуть сообщение в исходное положение, разделив его на $q/2$, чтобы устранить ошибку.

Сообщение равно 1, если оно ближе к $q/2$, чем к 0, в противном случае равно 0.

Разумеется, 1 бита недостаточно, поэтому для преодоления данного ограничения в современных схемах используются другие техники. На рис. 14.10 представлены все три алгоритма.

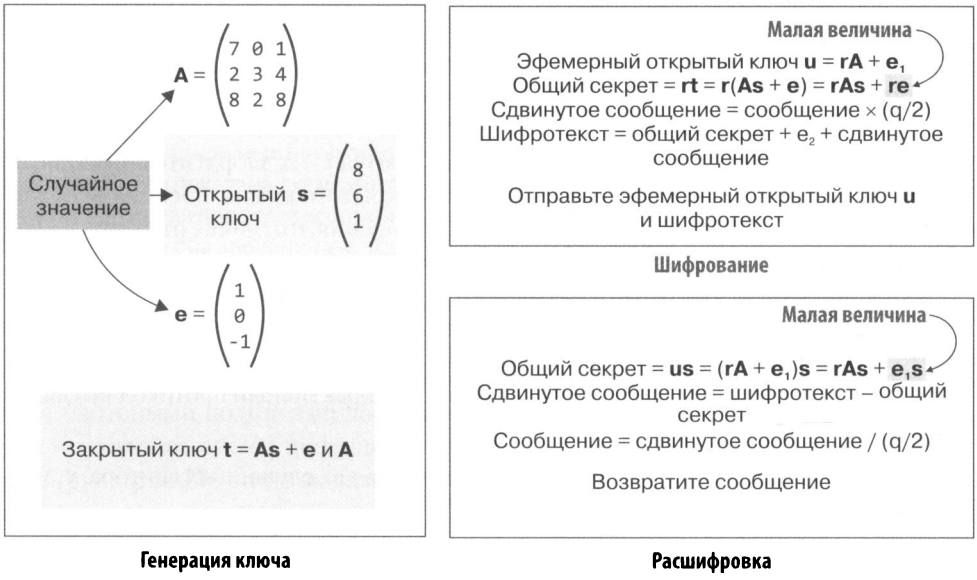


Рис. 14.10. Схема шифрования с открытым ключом Kyber. Обратите внимание на то, что общий секрет примерно одинаков при шифровании и расшифровке, так как re и e_1s — это малые величины, поскольку значения r, s и ошибки намного меньше q/2. Таким образом, на последнем шаге расшифровки (при делении на q/2, которое можно рассматривать как побитовый сдвиг вправо) мы избавляемся от любого несоответствия между двумя общими секретами. Внимание: все операции выполняются по модулю q

На практике при обмене ключами сообщение, которое вы зашифровываете открытым ключом собеседника, представляет собой случайный секрет. Затем результат детерминированным образом выводится из этого секрета и транскрипта обмена ключами, куда входят открытый ключ собеседника, ваш эфемерный ключ и шифротекст.

Из-за рекомендуемых параметров для схемы Kyber размер открытых ключей и шифротекстов составляет около 1 Кбайт, что гораздо больше по сравнению с применяемыми нами доквантовыми схемами, но вполне приемлемо для большинства случаев. Удастся ли сократить коммуникационные накладные расходы этих постквантовых схем, покажет время.

14.3.4. Dilithium — схема цифровой подписи на основе решетки

Следующая схема, о которой пойдет речь, *Dilithium*, также основана на задаче LWE и родственна схеме Kyber. Как и в случае с другими схемами цифровой подписи, которые мы уже рассмотрели, например со схемой подписи Шнорра (см. главу 7), Dilithium основана на доказательстве с нулевым разглашением, которое делается неинтерактивным с помощью трюка Фиата — Шамира.

При генерации ключа схема Dilithium работает аналогично предыдущей схеме, за исключением того, что ошибка является частью закрытого ключа. Сначала мы генерируем два случайных вектора $\mathbf{s}_1$ и $\mathbf{s}_2$, которые служат закрытым ключом, а затем вычисляем открытый ключ $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, где $\mathbf{A}$ — матрица, полученная тем же способом, что и в Kyber. Открытым ключом являются $\mathbf{t}$ и $\mathbf{A}$. Обратите внимание на то, что мы рассматриваем ошибку $\mathbf{s}_2$ как часть закрытого ключа, поскольку нужно использовать ее каждый раз при подписании сообщения, в отличие от схемы Kyber, позволяющей отбросить ошибку после этапа генерации ключа.

Для подписания мы создаем сигма-протокол, а затем преобразуем его в неинтерактивное доказательство с нулевым разглашением с помощью преобразования Фиата — Шамира, которое аналогично тому, как протокол идентификации Шнорра преобразуется в подпись Шнорра (см. главу 7). Интерактивный протокол выглядит следующим образом.

1. Доказывающий фиксирует в виде обязательства два случайных вектора, $\mathbf{y}_1$ и $\mathbf{y}_2$, отправляя $\mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$.
2. Получив это обязательство, проверяющий отвечает случайным вызовом c.
3. Затем доказывающий вычисляет два вектора, $\mathbf{z}_1 = c\mathbf{s}_1 + \mathbf{y}_1$ и $\mathbf{z}_2 = c\mathbf{s}_2 + \mathbf{y}_2$, и отправляет их проверяющему только в том случае, если они имеют малые значения.
4. Проверяющий выясняет, одинаковы ли значения $\mathbf{A}\mathbf{z}_1 + \mathbf{z}_2 - c\mathbf{t}$ и $\mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$.

Трюк Фиата — Шамира сводится к тому, что на шаге 2 вызов генерирует не проверяющий, а доказывающий на основе хеша подписываемого сообщения и фиксированного значения $\mathbf{A}\mathbf{y}_1 + \mathbf{y}_2$. Это изменение показано на рис. 14.11.

Данное описание является грубым упрощением схемы цифровой подписи. На практике для уменьшения размеров ключа и подписи используется гораздо больше оптимизаций. Обычно они направлены на уменьшение случайных значений путем их детерминированной генерации на основе меньшего случайного значения, а также на уменьшение объема неслучайных данных посредством их

сжатия с помощью специальных методов, причем не обязательно широко известных. Кроме того, существует ряд дополнительных оптимизаций, применение которых возможно благодаря уникальной структуре задачи LWE.



Рис. 14.11. Подпись Dilithium представляет собой доказательство знания секретного вектора s, сделанное неинтерактивным с помощью преобразования Фиата — Шамира. Слева показана схема интерактивного протокола доказательства, справа — неинтерактивная версия, в которой вызов вычисляется в виде обязательства, фиксирующего оба значения y и подписываемое сообщение

При рекомендуемом уровне безопасности схема Dilithium создает подписи размером около 3 Кбайт и открытые ключи размером менее 2 Кбайт. Очевидно, что это намного больше по сравнению с 32-байтовыми открытыми ключами и 64-байтовыми подписями доквантовых схем, но все-таки намного лучше того, что предлагают схемы цифровой подписи без сохранения состояния на основе хеш-функций. Не стоит забывать, что эти схемы относительно новые и, вполне вероятно, со временем будут разработаны более эффективные алгоритмы решения задачи LWE, что приведет к увеличению размеров открытых ключей и подписей. Возможно и то, что мы разработаем более совершенные техники уменьшения размеров этих параметров. В целом обеспечение квантовой устойчивости, скорее всего, всегда будет сопряжено с определенными издержками, обусловленными размерами параметров.

О постквантовой криптографии можно рассказать еще очень многое. На конкурсе NIST был представлен ряд других конструкций, основанных на различных парадигмах. Согласно заявлению NIST, первоначальный стандарт должен быть опубликован в 2022 году, но я ожидаю, что эта область будет очень быстро развиваться, по крайней мере пока квантовые компьютеры продолжают представлять угрозу. Тот факт, что в этой области существует много неизвестного, дает множество захватывающих возможностей для исследования. Если вас заинтересовала эта тема, рекомендую ознакомиться с отчетами NIST, доступными по адресу https://nist.gov/pqcrypto.

14.4. СТОИТ ЛИ ПАНИКОВАТЬ

Подводя итог, можно сказать, что реальное создание квантовых компьютеров будет иметь огромные последствия для криптографии. Какой вывод из этого следует? Стоит ли отказаться от того, что вы делаете сейчас, и перейти на постквантовые алгоритмы? Ну, все не так просто.

Если вы спросите разных экспертов, то получите разные ответы. Некоторые из них считают, что квантовые компьютеры могут быть созданы в ближайшие 5 или 50 лет, а другие думают, что они не будут созданы никогда. Мишель Моска, директор Института квантовых вычислений, оценил «вероятность взлома алгоритма RSA-2048 к 2026 году как 1/7, а к 2031 году — как 1/2», тогда как Михаил Дьяконов, исследователь из Национального центра научных исследований Франции (CNRS), заявил следующее: «Научимся ли мы когда-нибудь контролировать более чем 10^{300} постоянно изменяющихся параметров, определяющих квантовое состояние такой системы? Мой ответ прост: нет, никогда». Хотя физики разбираются в этом вопросе лучше, чем криптографы, у них может возникнуть соблазн приукрасить результаты своих исследований, чтобы получить финансирование. Поскольку я не физик, то просто скажу, что нам следует скептически относиться к чрезмерно оптимистичным заявлениям, готовясь при этом к худшему. Вопрос не в том, будет ли это работать, а в том, можно ли будет это масштабировать.

Для того чтобы масштабируемые квантовые компьютеры, способные уничтожить криптографию, стали реальностью, необходимо решить множество проблем, самые значительные из которых связаны с количеством шума и ошибок, что трудно уменьшить или исправить. Скотт Ааронсон, ученый-компьютерщик из Техасского университета, сказал по этому поводу: «Вы пытаетесь построить корабль, который остается тем же самым кораблем, даже когда все доски в нем гниют и подлежат замене».

Но как быть с заявлением АНБ? Нужно помнить, что потребность правительства в конфиденциальности чаще всего превышает потребности гражданских лиц и частных компаний. В том, что правительство может захотеть засекретить некоторые данные более чем на 50 лет, нет ничего безумного. Тем не менее заявление АНБ озадачило многих криптографов (см., например, работу Нила Коблица и Альфреда Дж. Менезеса *A Riddle Wrapped In An Enigma*), которые задались вопросом, зачем нам защищаться от того, чего еще не существует и, возможно, никогда не будет существовать.

В любом случае, если вы действительно обеспокоены и намерены обеспечивать конфиденциальность своих активов в течение длительного времени, вам имеет смысл увеличить параметры каждого симметричного криптографического алгоритма, который используете. Однако если вы выполняете обмен ключами для получения ключа AES-256-GCM, то имейте в виду, что эта часть асимметричной криптографии все еще уязвима для квантовых компьютеров, так что защиты одних лишь симметричных криптографических алгоритмов будет недостаточно.

Что касается асимметричной криптографии, то слишком рано говорить о том, что безопасно использовать, а что — нет. Лучше дождаться окончания конкурса NIST, чтобы получить больше результатов криптоанализа, а значит, и большую уверенность в новых алгоритмах.

> К настоящему моменту было предложено несколько постквантовых криптосистем, в том числе криптосистемы на решетках, криптосистемы на основе кодов, многомерные криптосистемы, схемы подписи на основе хеш-функций и некоторые другие. Однако для получения большей уверенности в безопасности большинства предложенных вариантов необходимо провести дальнейшие исследования (особенно с участием противников, обладающих квантовыми компьютерами) и повысить их производительность.
>
> *NIST Post-Quantum Cryptography Call for Proposals (2017)*

Если вы слишком нетерпеливы и не можете дождаться результатов конкурса NIST, попробуйте использовать в своем протоколе как текущую, так и постквантовую схему. Например, можете подписывать сообщения с помощью Ed25519 и Dilithium, или, другими словами, прикреплять к сообщению две подписи, созданные с помощью двух разных схем. Если схема Dilithium будет взломана, то злоумышленнику придется взломать и Ed25519, а если квантовые компьютеры все-таки будут созданы, то у вас останется подпись Dilithium, которую злоумышленник не сможет подделать.

ПРИМЕЧАНИЕ

В 2018-м, а затем и в 2019 году компания Google совместно с Cloudflare провела эксперимент с гибридной схемой обмена ключами в рамках TLS-соединений между небольшим процентом пользователей Google Chrome и серверами Google и Cloudflare. Эта схема представляла собой комбинацию алгоритма X25519 и одного из постквантовых алгоритмов обмена ключами (New Hope — в 2018-м, HRSS и SIKE — в 2019-м), выходные данные которых подавались на вход функции HKDF для получения единого общего секрета.

Наконец, стоит еще раз подчеркнуть, что схемы цифровой подписи на основе хеш-функций хорошо изучены и понятны. Несмотря на то что они сопряжены с некоторыми накладными расходами, схемы XMSS и SPHINCS+ можно использовать уже сейчас, а для XMSS даже существуют готовые стандарты (RFC 8391 и NIST SP 800-208).

РЕЗЮМЕ

- Квантовые компьютеры работают на основе явлений квантовой физики и могут существенно ускорять определенные вычисления.
- Не все алгоритмы могут выполняться на квантовом компьютере, и не все способны конкурировать с классическими компьютерами. Беспокойство криптографов вызывают два алгоритма:
  - алгоритм Шора, который позволяет эффективно решать задачи дискретного логарифмирования и факторизации, а значит, взламывать большинство современных асимметричных криптографических систем;
  - алгоритм Гровера, который может эффективно находить ключ или значение в пространстве из 2^{128} значений, а значит, взламывать большинство симметричных алгоритмов со 128-битным шифрованием. Чтобы защититься от подобных атак, достаточно увеличить параметры симметричного алгоритма для обеспечения 256-битного шифрования.

- Область постквантовой криптографии посвящена поиску новых криптографических алгоритмов, способных заменить современные асимметричные криптографические примитивы, например асимметричного шифрования, обмена ключами и цифровых подписей.
- В 2016 году организация NIST начала работу по стандартизации постквантовых криптографических алгоритмов. К настоящему моменту были выбраны семь финалистов, которые будут участвовать в заключительном раунде отбора.
- Подписи на основе хеш-функций — это схемы цифровой подписи, в основе которых лежат только хеш-функции и ничего больше. Два основных стандарта — XMSS (схема подписи с сохранением состояния) и SPHINCS+ (схема подписи без сохранения состояния).
- Криптография на решетках — весьма многообещающая область, так как позволяет создавать более короткие ключи и подписи. Двумя наиболее перспективными кандидатами, основанными на задаче LWE, являются Kyber (примитив асимметричного шифрования и обмена ключами) и Dilithium (схема цифровой подписи).
- В рамках проводимого NIST конкурса предлагаются и другие постквантовые схемы, в том числе основанные на теории кодов, изогений, криптографии с симметричным ключом и многомерных полиномов. Конкурс NIST должен завершиться в 2022 году, что оставляет множество возможностей для разработки новых атак и оптимизаций.
- Пока неясно, когда квантовые компьютеры станут достаточно эффективными для того, чтобы уничтожить криптографию, и способны ли они на это в принципе.
- Если вы хотите обеспечить долгосрочную защиту своих данных, рассмотрите возможность перехода на постквантовые криптографические алгоритмы.
  - Увеличьте параметры используемых симметричных криптографических алгоритмов, чтобы обеспечить 256-битное шифрование (например, перейдите с AES-128-GCM на AES-256-GCM и с SHA-3-256 — на SHA-3-512).
  - Используйте гибридные схемы, сочетающие доквантовые и постквантовые алгоритмы. Например, всегда подписывайте сообщения с помощью Ed25519 и Dilithium или выполняйте обмен ключами посредством X25519 и Kyber, генерируя общий секрет на основе двух результатов обмена ключами.
  - Задействуйте схемы цифровой подписи на основе хеш-функций, такие как XMSS и SPHINCS+, уже хорошо изученные и понятные. Преимущество схемы XMSS заключается в том, что она уже стандартизирована и одобрена организацией NIST.

15 Криптография следующего поколения

В этой главе

- ✓ Избавление от доверенных третьих лиц с помощью защищенного многостороннего вычисления (MPC).
- ✓ Предоставление другим лицам возможности работать с зашифрованными данными с помощью полностью гомоморфного шифрования (FHE).
- ✓ Сокрытие частей выполняющейся программы с помощью доказательств с нулевым разглашением (ZKP).

Я начал писать эту книгу с мыслью о том, что читателей, преодолевших большинство ее глав, также интересует будущее криптографии. Хотя книга является прикладной и посвящена тем методам, которые используются сегодня, область криптографии быстро меняется, как мы видели в последние годы на примере криптовалют.

Пока вы читаете, ряд теоретических криптографических примитивов и протоколов пробивают себе дорогу в мир прикладной криптографии. Это может быть связано с тем, что данные теоретические примитивы наконец-то нашли применение, или с тем, что они наконец-то стали достаточно эффективными для использования в реальных приложениях. Какой бы ни была причина, сфера реальной криптографии определенно развивается и становится все более интересной. В этой главе я дам представление о том, как может измениться мир

реальной криптографии, например, в ближайшие 10–20 лет благодаря внедрению следующих трех примитивов:

- *защищенного многостороннего вычисления* (secure multi-party computation, MPC). Так называется область криптографии, позволяющая разным участникам выполнять программу совместно, не раскрывая при этом собственные входные данные;
- *полностью гомоморфного шифрования* (fully homomorphic encryption, FHE). Это святой Грааль криптографии, примитив, позволяющий выполнять произвольные вычислительные операции над зашифрованными данными;
- *многоцелевых доказательств с нулевым разглашением* (general-purpose zero-knowledge proof, ZKP). Это примитив, описанный в главе 7, который позволяет доказать, что вы знаете нечто, не раскрывая этого, только на этот раз применяющийся в более общем виде и к более сложным программам.

В данной главе представлены наиболее продвинутые и сложные концепции. По этой причине рекомендую ознакомиться с ней бегло, а затем перейти к главе 16, чтобы прочитать заключение. Когда у вас появится желание разобраться в сути этих более сложных концепций, вернитесь к данной главе. Итак, начнем!

15.1. ЧЕМ БОЛЬШЕ НАРОДУ, ТЕМ ВЕСЕЛЕЕ: ЗАЩИЩЕННЫЕ МНОГОСТОРОННИЕ ВЫЧИСЛЕНИЯ

Защищенные, или безопасные, многосторонние вычисления (MPC) — это область криптографии, которая возникла в 1982 году благодаря знаменитой задаче миллионеров, предложенной Эндрю К. Яо в работе *Protocols for Secure Computations*. Ее суть сводится к тому, что два миллионера хотят выяснить, кто из них богаче, не желая при этом случайно узнать какую-либо дополнительную информацию о состояниях друг друга. Как они могут это сделать? Если не усложнять, то MPC представляет собой способ распределения вычислений между несколькими участниками. Но прежде, чем узнать больше о новом примитиве, давайте разберемся, чем он может быть полезен.

Мы знаем, что любые распределенные вычисления можно легко произвести с помощью доверенной третьей стороны, способной обеспечить конфиденциальность входных данных каждого участника, а также ограничить количество информации, получаемой в ходе вычислений конкретными участниками. Однако в реальном мире нам не слишком нравится полагаться на доверенную третью сторону, поскольку ее довольно трудно найти и она не всегда соблюдает взятые на себя обязательства.

Концепция MPC позволяет полностью исключить участие доверенной третьей стороны из процесса распределенных вычислений и дает участникам возможность выполнять их самостоятельно, не раскрывая при этом свои входные данные. Учитывая то, что это делается с помощью криптографического протокола, использование MPC в системе практически эквивалентно участию в процессе доверенной третьей стороны (рис. 15.1).

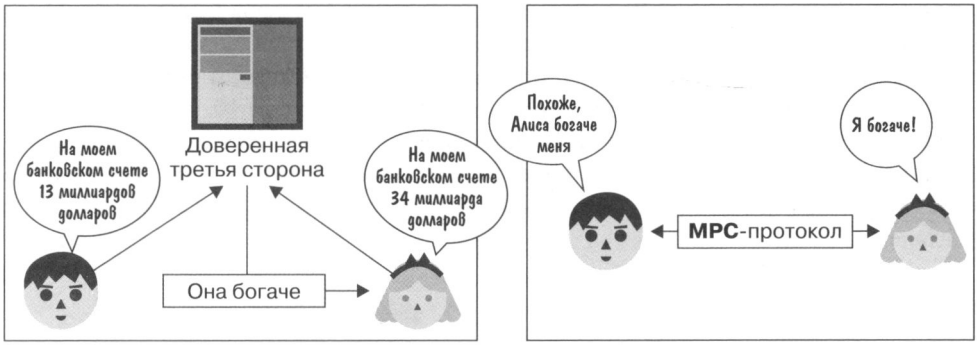


Рис. 15.1. Протокол защищенных многосторонних вычислений превращает распределенные вычисления, требующие участия доверенной третьей стороны *(слева)*, в вычисления, которые этого не требуют *(справа)*

С некоторыми протоколами MPC вы уже знакомы. Их пример — пороговая подпись и распределенная генерация ключей, о которых шла речь в главе 8. Если говорить более конкретно, то эти примеры относятся к области MPC, называемой *пороговой криптографией*, которая в последнее время пользуется большим вниманием, о чем свидетельствует процесс стандартизации, начатый в середине 2019 года организацией NIST.

15.1.1. Пересечение закрытых множеств

Другая известная область MPC называется *пересечением закрытых множеств* (private set intersection, PSI) и ставит следующую задачу: у Алисы и Боба есть списки слов и они хотят узнать, какие из них общие, не раскрывая свой список (или просто выяснить количество таких слов). Один из способов решения этой задачи предполагает использование забывчивой псевдослучайной функции (OPFR), о которой мы говорили в главе 11. (Схема данного протокола проиллюстрирована на рис. 15.2.) Как вы помните, происходит следующее.

1. Боб генерирует ключ для OPRF.
2. Алиса получает случайные значения, *PRF*(*ключ*, *слово*), для каждого слова своего списка, применяя протокол OPRF (таким образом она не узнает значение ключа PRF, а Боб не узнает слова).
3. Затем Боб может вычислить список значений *PRF*(*ключ*, *слово*) для своих слов и отправить его Алисе, которая сравнит его с собственными PRF-выходами на предмет совпадения.

PSI — это весьма многообещающая область, которая за последние годы получила довольно широкое распространение благодаря возросшей практичности. Например, встроенная в браузер Chrome функция Google Password Checkup использует PSI, чтобы предупредить вас о том, что некоторые из ваших паролей были обнаружены в базах украденных учетных данных, не видя при этом реальных

паролей. Компания Microsoft разработала для своего браузера Edge аналогичную функцию, но применяла для этого полностью гомоморфное шифрование (о нем я расскажу в следующем разделе). В то же время разработчики приложения для обмена сообщениями Signal, о котором мы говорили в главе 10, решили, что PSI обеспечивает слишком медленное обнаружение в вашем списке контактов тех людей, с которыми вы можете общаться в этом мессенджере, и поэтому вместо PSI использовали в качестве доверенной третьей стороны среду SGX, упомянутую в главе 13.

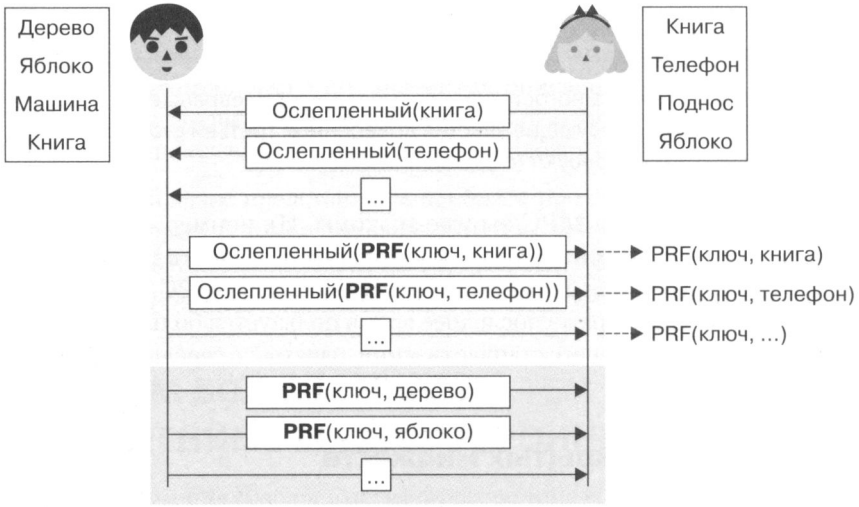


Рис. 15.2. Пересечение закрытых множеств позволяет Алисе узнать, какие слова из ее списка совпадают со словами из списка Боба. Сначала она ослепляет каждое слово из своего списка и использует OPRF-протокол вместе с Бобом, чтобы применить к каждому из своих слов функцию PRF с помощью ключа Боба. Наконец, Боб отправляет ей PRF своего ключа вместе со своими словами, после чего Алиса может сравнить его слова со своими на предмет совпадения

15.1.2. MPC общего назначения

Существует множество MPC-решений, нацеленных на выполнение произвольных вычислений. Подобные MPC-решения общего назначения предусматривают различные уровни эффективности (от часов до миллисекунд) и типы свойств. Например, сколько нечестных участников может выдержать протокол? Являются ли участники злонамеренными или честными, но любопытными? (Таких участников MPC-протоколов называют также *получестными*, поскольку они готовы выполнять протокол правильно, но могут и попытаться узнать входные данные других участников.) Если некоторые из участников завершат выполнение протокола раньше времени, будет ли это справедливо по отношению к остальным?

Перед выполнением вычисления с помощью MPC программа должна быть преобразована в *арифметическую схему*. Арифметические схемы (рис. 15.3) представляют собой последовательности операций сложения и умножения, и поскольку они являются полными по Тьюрингу, то могут представлять *любую* программу!

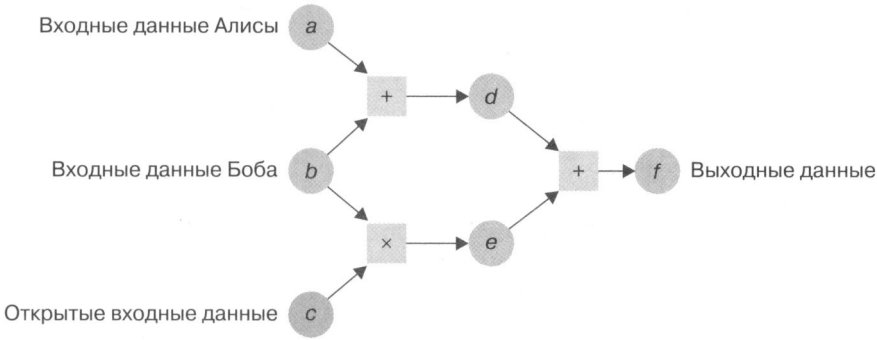


Рис. 15.3. Арифметическая схема — это последовательность операций сложения или умножения, соединяющих входы с выходами. На представленном изображении значения перемещаются слева направо, например d = a + b. В данном случае схема выдает только одно значение f = a + b + bc, однако теоретически она может иметь несколько выходных значений. Обратите внимание на то, что в схеме используются входные данные, предоставляемые разными участниками, однако эти данные могут быть и открытыми, то есть известными всем

Прежде чем мы перейдем к рассмотрению следующего примитива, позвольте мне привести упрощенный пример MPC-решения общего назначения (с большинством честных участников), построенного на основе схемы разделения секрета Шамира. Существует множество других схем, но эта достаточно проста для того, чтобы ее можно было представить в виде последовательности из трех шагов: предоставление достаточного количества информации о каждом входе схемы, выполнение операций, определяемых ее вентилями, и восстановление выхода. Рассмотрим каждый из шагов подробнее.

Первый шаг заключается в том, чтобы предоставить каждому участнику достаточное количество информации о каждом входе схемы. Открытые входы доступны всем, а закрытые распределяются с помощью схемы разделения секрета Шамира (см. главу 8). Я проиллюстрировал этот процесс на рис. 15.4.

На втором шаге выполняются операции, определяемые вентилями схемы. По техническим причинам, на которых я не стану здесь останавливаться, операции сложения могут выполняться локально, тогда как операции умножения должны выполняться интерактивно, то есть участники должны обмениваться некоторыми сообщениями. Для выполнения операции сложения просто сложите имеющиеся у вас доли входов, для выполнения операции умножения — перемножьте их. В итоге вы получите некоторую долю результата (рис. 15.5). На этом этапе долями можно обменяться (чтобы восстановить выход) или продолжить вычисления (если доли представляют собой промежуточные значения).

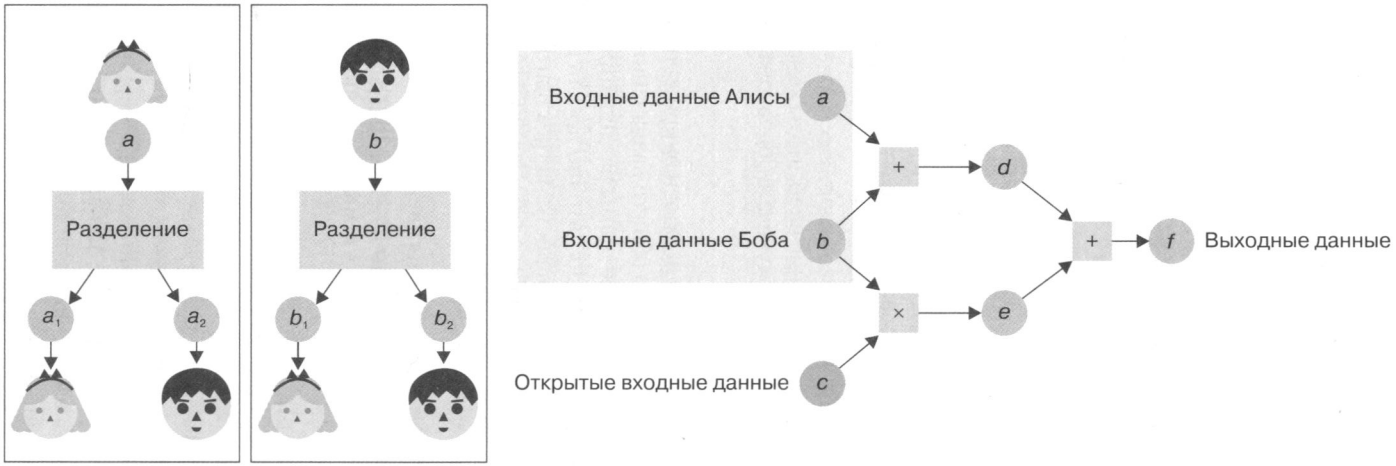


Рис. 15.4. Первый шаг схемы MPC общего назначения с разделением секрета сводится к тому, что участники разделяют свои секретные входы с помощью схемы разделения секрета Шамира и распределяют их доли между всеми участниками. Например, в данном случае Алиса разделяет свой вход a на a_1 и a_2. Поскольку здесь только два участника, она оставляет первую долю себе, а вторую отдает Бобу

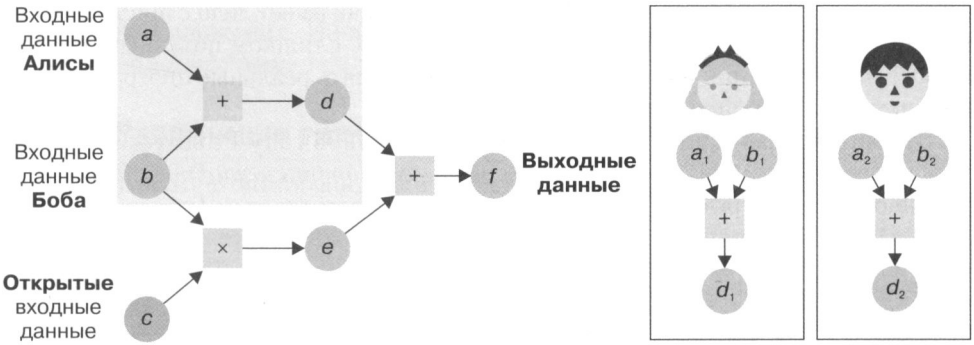


Рис. 15.5. Второй шаг схемы MPC общего назначения с разделением секрета заключается в выполнении операций, определяемых вентилями схемы. Например, участник может выполнить операцию сложения путем суммирования двух имеющихся у него долей входов, чтобы получить долю выхода

Последний шаг предполагает восстановление выходных данных. На этом этапе все участники должны владеть долей выхода, которую они могут использовать для восстановления конечного результата с помощью последнего шага схемы разделения секрета Шамира.

15.1.3. Текущее состояние дел в сфере MPC

За последнее десятилетие для повышения практичности многосторонних вычислений было сделано очень много. Эта область постоянно пополняется вариантами приложения, позволяющими извлечь выгоду из использования этого нового примитива. Однако обратите внимание на то, что реальных усилий по стандартизации пока не предпринимается и, хотя некоторые реализации MPC уже считаются пригодными для многих ситуаций, использовать их нелегко.

Хотя схема MPC общего назначения, о которой я рассказывал ранее в этом разделе, основана на разделении секрета, существуют и другие способы построения MPC-протоколов. Хорошо известная альтернатива — так называемые *искаженные схемы* (garbled circuits), представляющие собой тип конструкции, впервые предложенный Яо в работе 1982 года, в которой он представил концепцию MPC. Другая альтернатива основана на полностью гомоморфном шифровании — примитиве, речь о котором пойдет в следующем разделе.

15.2. ПОЛНОСТЬЮ ГОМОМОРФНОЕ ШИФРОВАНИЕ И ОБЕЩАНИЕ ЗАШИФРОВАННОГО ОБЛАКА

Многим криптографам долго не давал покоя вопрос: можно ли выполнять произвольные вычислительные операции над зашифрованными данными? Представьте, что у вас есть возможность по отдельности зашифровать значения a, b и c, отправить шифротексты на сервис и попросить его возвратить результат вычисления $a \times 3b + 2c + 3$, который вы затем сможете расшифровать. Важно здесь то, что этот

сервис никогда не узнает истинных значений, так как имеет дело с шифротекстами. Приведенное вычисление может показаться не слишком полезным, однако из операций сложения и умножения можно составлять реальные программы, выполняемые на зашифрованных данных.

Эта интересная концепция, предложенная в 1978 году Ривестом, Адлеманом и Дертузосом, называется *полностью гомоморфным шифрованием* (fully homomorphic encryption, FHE) (раньше его называли *святым Граалем криптографии*). Данный криптографический примитив проиллюстрирован на рис. 15.6.

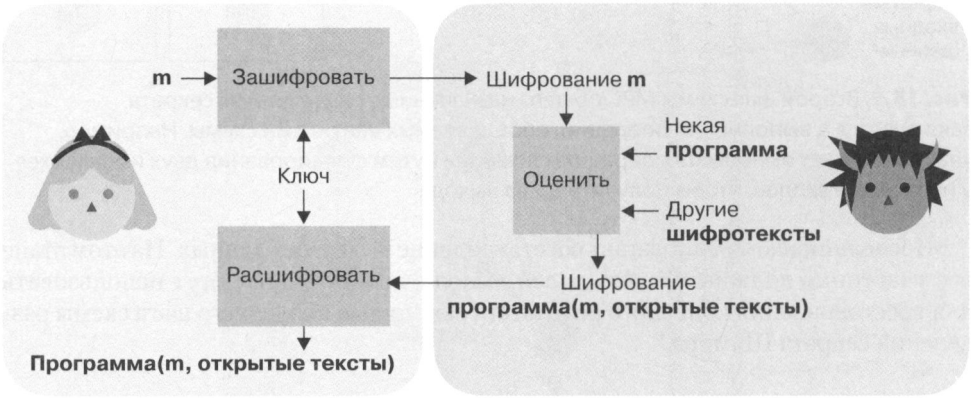


Рис. 15.6. Полностью гомоморфное шифрование — это схема шифрования, которая позволяет выполнять произвольные вычислительные операции над зашифрованным содержимым. Расшифровать результат этих вычислений может только владелец ключа

15.2.1. Пример гомоморфного шифрования с помощью алгоритма RSA

Кстати, вы уже познакомились с некоторыми криптографическими схемами, которые должны облегчить понимание того, о чем я говорю. Вспомните алгоритм RSA (описанный в главе 6): при заданном значении *шифротекст* = *сообщение*^{e} mod N кто-то может легко вычислить некоторую ограниченную функцию шифротекста:

$$n^e \times \textit{шифротекст} = (n \times \textit{сообщение})^e \bmod N -$$

для любого числа n (хотя оно не может быть слишком большим). В результате получается шифротекст, который расшифровывается следующим образом:

$$n \times \textit{сообщение}.$$

Разумеется, такое поведение алгоритма RSA нежелательно, так как делает возможным реализацию некоторых атак, в частности атаки Блейхенбахера, о которой говорилось в главе 6. На практике алгоритм RSA нарушает свойство гомоморфизма, используя схему дополнения. Обратите внимание на то, что криптосистема RSA гомоморфна только относительно операций умножения, чего оказывается

недостаточно для вычисления произвольных функций, которое требует как умножения, так и сложения. Из-за этого ограничения мы называем алгоритм RSA *частично гомоморфным*.

15.2.2. Различные типы гомоморфного шифрования

Другими типами гомоморфного шифрования являются следующие.

- *В некоторой степени гомоморфное шифрование* означает, что криптосистема частично гомоморфна относительно одной операции (сложения или умножения) и гомоморфна относительно другой в определенных пределах. Например, операции сложения могут выполняться неограниченно вплоть до определенного предела, а операция умножения может быть выполнена всего несколько раз.
- *Ограниченно гомоморфное шифрование* позволяет выполнять как сложение, так и умножение определенное количество раз.
- *Полностью гомоморфное шифрование* предполагает неограниченное количество операций сложения и умножения.

До изобретения технологии FHE было предложено несколько типов схем гомоморфного шифрования, но ни одна из них не обеспечила того, что обещало полностью гомоморфное шифрование. Причина в том, что при выполнении вычислительных операций над зашифрованными данными количество *шума* возрастает и в какой-то момент достигает предельного значения, делающего расшифровку невозможной. И на протяжении многих лет некоторые исследователи пытались сформулировать теорию информации, которая доказывала бы невозможность реализации полностью гомоморфного шифрования, однако в конце концов было показано, что она все-таки возможна.

15.2.3. Перешифрование — ключ к полностью гомоморфному шифрованию

> Однажды ночью Алиса видит во сне огромные богатства — пещеры, заполненные серебром, золотом и бриллиантами. Затем гигантский дракон пожирает все эти сокровища, после чего принимается за собственный хвост! Она просыпается с ощущением покоя. Пытаясь осмыслить свой сон, она понимает, что нашла решение проблемы.
>
> *Крейг Джентри (Computing Arbitrary Functions of Encrypted Data, 2009)*

В 2009 году Крейг Джентри, аспирант Дэна Боне, предложил первую в истории систему полностью гомоморфного шифрования. Свое решение он назвал *перешифрованием* (bootstrapping), суть его заключается в повторном шифровании данных после некоторого количества операций с целью снижения уровня шума до приемлемого порога. Интересно то, что сама схема расшифровки не раскрывает закрытый ключ и может вычисляться недоверенной стороной. Перешифрование позволило превратить схему ограниченно гомоморфного шифрования в схему полностью гомоморфного шифрования. Система Джентри была медленной

и довольно непрактичной: на выполнение одной базовой битовой операции уходило около 30 мин, однако, как и любая другая прорывная технология, со временем она становилась только лучше. А еще она показала, что полностью гомоморфное шифрование возможно.

Как же работает перешифрование? Попробуем в этом разобраться. Вначале следует отметить, что для его выполнения нам понадобится не симметричная система шифрования, а система шифрования с открытым ключом, причем открытый ключ будет использоваться для шифрования, а закрытый — для расшифровки. Теперь представьте, что после выполнения определенного количества операций сложения и умножения над шифротекстом уровень шума достигает порогового значения, при котором вы все еще можете правильно расшифровать шифротекст, но не можете выполнять дальнейшие гомоморфные операции, не уничтожив при этом зашифрованное содержимое (рис. 15.7).

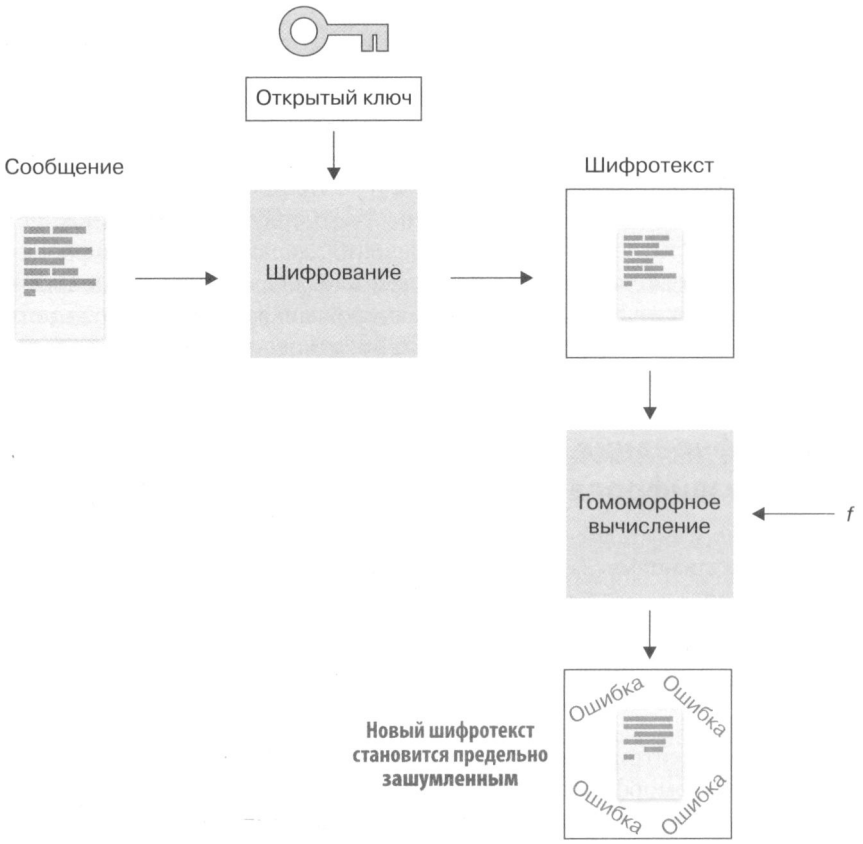


Рис. 15.7. После шифрования сообщения с помощью полностью гомоморфного алгоритма выполнение операций над ним приводит к увеличению уровня шума до опасных пороговых значений, при которых расшифровка становится невозможной

Вам может показаться, что вы столкнулись с непреодолимой проблемой, но перешифрование позволяет справиться с ней, удаляя шум из шифротекста. Для этого вы повторно шифруете зашумленный шифротекст с использованием другого открытого ключа, обычно называемого *ключом перешифрования* (bootstrapping key), чтобы получить шифр этого зашумленного шифротекста. Обратите внимание на то, что новый шифротекст не содержит шума (рис. 15.8).

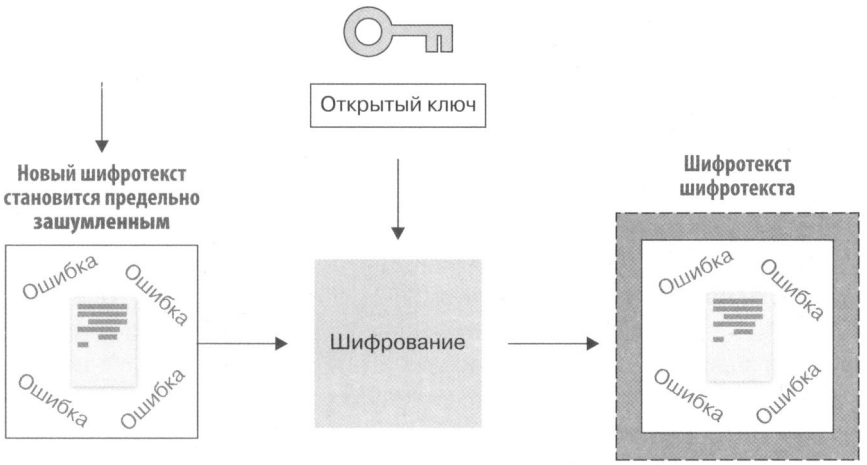


Рис. 15.8. Дополнение к схеме, приведенной на рис. 15.7. Чтобы устранить шум из шифротекста, можете расшифровать его. Однако поскольку у вас нет секретного ключа, вместо этого вы повторно шифруете зашумленный шифротекст с использованием другого открытого ключа, называемого ключом перешифрования, чтобы получить новый шифротекст зашумленного шифротекста, не содержащий ошибок

Теперь начинается волшебство: вам предоставляется исходный закрытый ключ, но не в открытом виде, а зашифрованный с помощью ключа перешифрования. Это означает, что вы можете использовать его для гомоморфной расшифровки зашумленного шифротекста. Если схема расшифровки производит приемлемое количество шума, то она работает, и в итоге вы получите результат первой гомоморфной операции, зашифрованный с помощью ключа перешифрования (рис. 15.9).

Если оставшееся количество ошибок позволяет выполнить хотя бы еще одну гомоморфную операцию, сложение (+) или умножение (×), то у вас имеется полностью гомоморфный алгоритм шифрования, поскольку на практике вы можете выполнять перешифрование после каждой операции или перед ней. Обратите внимание на то, что можно сделать так, чтобы пара ключей перешифрования совпадала с парой исходных ключей. Это немного странно, но, похоже, работает и не вызывает никаких проблем с безопасностью.

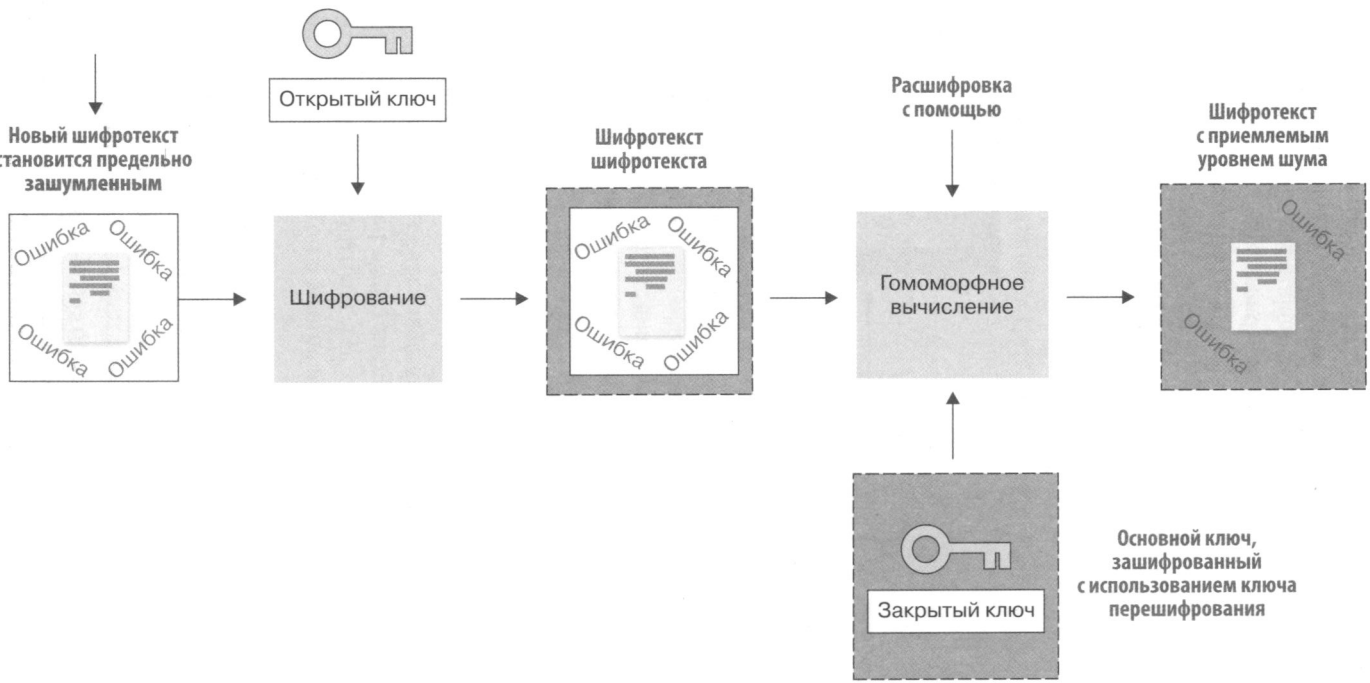


Рис. 15.9. Дополнение к схеме, приведенной на рис. 15.8. Вы используете исходный секретный ключ, зашифрованный с помощью ключа перешифрования, чтобы применить схему расшифровки к новому шифротексту. При этом зашумленный шифротекст расшифровывается на месте, очищаясь от ошибок. Оставшееся количество ошибок объясняется работой схемы расшифровки

15.2.4. Схема полностью гомоморфного шифрования, основанная на задаче обучения с ошибками

Прежде чем двигаться дальше, рассмотрим пример схемы полностью гомоморфного шифрования (FHE), основанной на задаче обучения с ошибками (LWE), о которой говорилось в главе 14. Далее я представлю упрощенную версию схемы GSW, названной так в честь ее авторов Крейга Джентри, Амита Сахаи и Брента Уотерса. Чтобы не усложнять, я опишу версию алгоритма с закрытым ключом, однако имейте в виду, что такую схему довольно легко преобразовать в вариант с открытым ключом, который нужен для выполнения перешифрования. Взгляните на следующее уравнение, где $\mathbf{C}$ — это квадратная матрица, $\mathbf{s}$ — вектор, а m — скаляр (число):

$$\mathbf{Cs} = m\mathbf{s}.$$

В этом уравнении $\mathbf{s}$ называется *собственным вектором*, а m — *собственным значением*. Если эти слова вам незнакомы, не волнуйтесь, они не имеют большого значения для последующего объяснения.

При взгляде на собственные векторы и собственные значения нашей FHE-схемы становится ясно, что если m — это единственный бит, который мы хотим зашифровать, $\mathbf{C}$ — шифротекст, а $\mathbf{s}$ — секретный ключ, то это небезопасная схема гомоморфного шифрования одного бита. (Разумеется, мы предполагаем, что существует способ получить случайный шифротекст $\mathbf{C}$ из фиксированного бита m и фиксированного секретного ключа $\mathbf{s}$.) Я проиллюстрировал это на рис. 15.10.

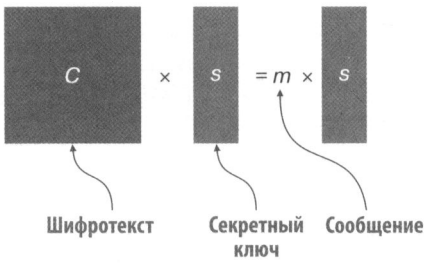


Рис. 15.10. Можно создать небезопасную схему гомоморфного шифрования для шифрования одного бита m с помощью секретного вектора s, интерпретируя m в качестве собственного значения, s — в качестве собственного вектора, а затем найдя связанную матрицу C, которая и будет шифротекстом

Чтобы расшифровать шифротекст, умножаем матрицу на секретный вектор $\mathbf{s}$ и смотрим, что получается в итоге — секретный вектор или 0. Можно убедиться в том, что схема полностью гомоморфна, проверив, приводит ли расшифровка двух сложенных шифротекстов ($\mathbf{C}_1 + \mathbf{C}_2$) к получению суммы соответствующих битов:

$$(\mathbf{C}_1 + \mathbf{C}_2)\mathbf{s} = \mathbf{C}_1\mathbf{s} + \mathbf{C}_2\mathbf{s} = b_1\mathbf{s} + b_2\mathbf{s} = (b_1 + b_2)\mathbf{s}.$$

Результатом расшифровки двух перемноженных шифротекстов ($\mathbf{C}_1\mathbf{C}_2$) также является произведение соответствующих битов:

$$(\mathbf{C}_1\mathbf{C}_2)\mathbf{s} = \mathbf{C}_1(\mathbf{C}_2\mathbf{s}) = \mathbf{C}_1(b_2\mathbf{s}) = b_2\mathbf{C}_1\mathbf{s} = (b_2b_1)\mathbf{s}.$$

К сожалению, эта схема небезопасна, так как из $\mathbf{C}$ можно легко извлечь собственный вектор — секретный вектор $\mathbf{s}$. А что, если добавить немного шума? Можем слегка изменить это уравнение, сделав его похожим на задачу обучения с ошибками:

$$\mathbf{Cs} = m\mathbf{s} + \mathbf{e}.$$

Это выражение должно показаться вам знакомым. И снова можем убедиться в том, что операция сложения по-прежнему гомоморфна:

$$(\mathbf{C}_1 + \mathbf{C}_2)\mathbf{s} = \mathbf{C}_1\mathbf{s} + \mathbf{C}_2\mathbf{s} = b_1\mathbf{s} + \mathbf{e}_1 + b_2\mathbf{s} + \mathbf{e}_2 = (b_1 + b_2)\mathbf{s} + (\mathbf{e}_1 + \mathbf{e}_2).$$

Обратите внимание на то, что ошибка растет ($\mathbf{e}_1 + \mathbf{e}_2$), чего мы и ожидали. Можем также убедиться в том, что умножение по-прежнему работает:

$$(\mathbf{C}_1\mathbf{C}_2)\mathbf{s} = \mathbf{C}_1(\mathbf{C}_2\mathbf{s}) = \mathbf{C}_1(b_2\mathbf{s} + \mathbf{e}_2) = b_2\mathbf{C}_1\mathbf{s} + \mathbf{C}_1\mathbf{e}_2 = b_2(b_1\mathbf{s} + \mathbf{e}_1) + \mathbf{C}_1\mathbf{e}_2 =$$

$$= (b_2 \times b_1)\,\mathbf{s} + b_2\mathbf{e}_1 + \mathbf{C}_1\mathbf{e}_2.$$

Здесь значение $b_2\mathbf{e}_1$ маленькое, поскольку оно равно либо $\mathbf{e}_1$, либо 0, а значение $\mathbf{C}_1\mathbf{e}_2$ может быть довольно большим. Очевидно, что это проблема, но я ее проигнорирую, чтобы не сильно углубляться в детали. Если хотите узнать больше, ознакомьтесь с докладом Шая Халеви *Homomorphic Encryption* (2017), в котором отлично объясняются все эти и многие другие вещи.

15.2.5. Сфера применения полностью гомоморфного шифрования

Самый популярный вариант использования FHE всегда был связан с облачной технологией. Что, если бы у нас была возможность хранить свои данные в облаке так, чтобы их никто не мог просмотреть? И что, если бы облако могло выполнять полезные вычисления над этими зашифрованными данными? Действительно, для FHE можно придумать множество способов применения. Вот несколько примеров.

- Детектор спама может просканировать ваши письма, не узнавая их содержания.
- Генетические исследования вашей ДНК могут проводиться, не требуя хранения и защиты конфиденциальной информации.
- База данных может храниться в зашифрованном виде и запрашиваться на стороне сервера без раскрытия каких-либо сведений.

Однако в своей значимой работе 2015 года *The Moral Character of Cryptographic Work* Филипп Рогауэй отметил, что «технология FHE <...> породила очередную волну оптимизма. В заявках на получение грантов, в интервью и выступлениях ведущие теоретики говорят о FHE <...> как о чем-то революционном. При этом никто не подчеркивает, насколько далеко все это от реализации на практике».

Хотя Рогауэй прав в том смысле, что FHE все еще работает довольно медленно, в этой области наблюдается существенный прогресс. На момент написания книги (2021 год) подобные операции выполнялись примерно в миллиард раз медленнее по сравнению с обычными, однако с 2009-го их скорость увеличилась в 10^9 раз. Нет никаких сомнений в том, что в будущем полностью гомоморфное шифрование будет применяться хотя бы в некотором ограниченном ряде случаев.

Кроме того, не каждый такой случай требует полноценного примитива — частично гомоморфное шифрование, гораздо более эффективное по сравнению с FHE, тоже может использоваться в широком спектре приложений. Хорошим показателем того, что теоретический криптографический примитив находит применение в реальном мире, является стандартизация, которая не обошла стороной и технологию FHE (https://homomorphicencryption.org). В этой работе участвуют многие крупные компании и университеты. Пока точно не известно, когда, где и в какой форме гомоморфное шифрование проявит себя в реальном мире. Ясно лишь то, что это обязательно произойдет, так что следите за новостями!

15.3. МНОГОЦЕЛЕВЫЕ ДОКАЗАТЕЛЬСТВА С НУЛЕВЫМ РАЗГЛАШЕНИЕМ

О доказательствах с нулевым разглашением (ZKP) мы говорили в главе 7, посвященной цифровым подписям. Там я отметил, что подписи похожи на неинтерактивные ZKP-знания для дискретных логарифмов. Эти ZKP были изобретены в середине 1980-х годов профессорами Шафи Голдвассером, Сильвио Микали и Чарльзом Рэкоффом. Вскоре после этого Голдрейх, Микали и Вигдерсон обнаружили, что, помимо дискретных логарифмов и других сложных задач, мы можем доказать правильность выполнения любой программы даже при удалении некоторых входных или выходных данных (пример показан на рис. 15.11). Данный раздел посвящен именно этому типу многоцелевых ZKP.

Область ZKP сильно эволюционировала с момента возникновения. Одними из основных причин роста стали криптовалютный бум и необходимость в обеспечении большей конфиденциальности транзакций в блокчейне, а также оптимизации использования пространства. Сейчас данная область по-прежнему стремительно развивается, поэтому уследить за всеми современными схемами и типами ZKP довольно сложно.

К счастью для нас, эта проблема стала достаточно большой для того, чтобы перешагнуть *порог стандартизации* — воображаемую черту, которая, будучи достигнутой, почти всегда побуждает множество людей к совместной работе над внесением уточнений в ту или иную область. В 2018 году представители индустрии и научных кругов объединились для реализации проекта ZKProof Standardization с целью «стандартизировать использование криптографических доказательств с нулевым разглашением». И эта работа продолжается по сей день. Подробнее о ней можно узнать на сайте https://zkproof.org.

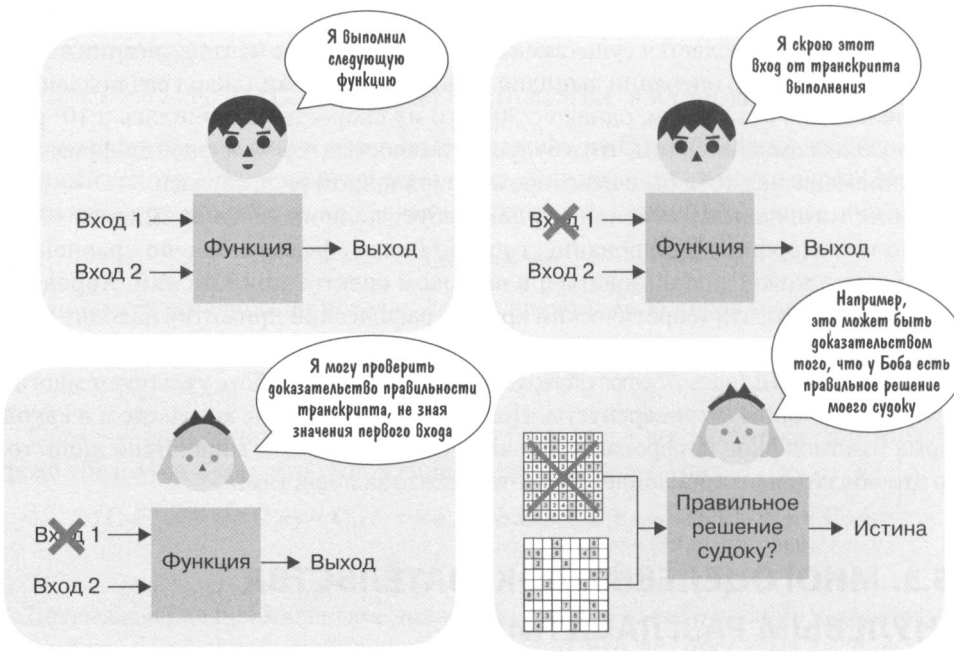


Рис. 15.11. Многоцелевые ZKP позволяют доказывающему убедить проверяющего в целостности трассы выполнения (входов программы и выходов, полученных после ее реализации), скрывая при этом некоторые из входов или выходов, задействованных в вычислениях. В качестве примера можно привести того, кто пытается доказать возможность решения судоку

Многоцелевые ZKP можно применять в самых разных ситуациях, но, насколько мне известно, до сих пор они применялись в основном в криптовалютной сфере. Вероятно, это объясняется большим количеством людей, интересующихся криптографией и желающих поэкспериментировать с передовыми технологиями. Тем не менее многоцелевые ZKP могут найти применение во многих областях, включая управление идентификацией (возможность подтвердить свой возраст, не раскрывая его), сжатие (возможность скрыть большую часть вычислений), конфиденциальность (возможность скрыть части протокола) и т. д. Самыми большими препятствиями для внедрения многоцелевых ZKP являются:

- постоянно возрастающее количество ZKP-схем;
- сложность понимания принципа работы этих систем и того, как их можно использовать в конкретных случаях.

Различия между предложенными схемами очень важны, поскольку они оказываются существенным источником путаницы. Вот по каким критериям обычно различаются эти схемы.

- *Степень секретности.* Если часть информации должна держаться в секрете от некоторых участников, то мы говорим о нулевом разглашении. Обратите внима-

ние на то, что доказательства без секретов тоже могут быть полезны. Например, вы можете делегировать некоторые интенсивные вычисления сервису, который, в свою очередь, должен доказать вам правильность выданного им результата.

- *Интерактивность.* Большинство ZKP-схем можно сделать неинтерактивными (иногда с помощью преобразования Фиата — Шамира, о котором мы говорили в главе 7), и похоже, что разработчики протоколов чаще всего предпочитают неинтерактивные версии схемы. Это связано с тем, что обмен сообщениями в рамках протокола может занимать много времени, к тому же иногда обеспечить интерактивность невозможно. Так называемые неинтерактивные доказательства часто обозначают аббревиатурой *NIZK* (от non-interactive ZKP).
- *Лаконичность доказательства.* Большинство ZKP-схем, находящихся в центре внимания, часто обозначают аббревиатурой *zk-SNARK* (zero-knowledge succinct non-interactive argument of knowledge — краткий неинтерактивный аргумент знания с нулевым разглашением). Хотя определение может варьироваться, обычно в нем делается акцент на размере доказательств, производимых подобными системами (как правило, он измеряется сотнями байт), и на количестве времени, необходимого для их проверки (обычно на это уходит несколько миллисекунд). Таким образом, схемы zk-SNARK позволяют быстро и легко проверить ZKP. Следует отметить: то, что схема не принадлежит к категории zk-SNARK, не исключает возможности ее применения в реальном мире, поскольку в разных ситуациях могут пригодиться разные свойства.
- *Степень прозрачности.* Как и любой другой криптографический примитив, ZKP нуждаются в настройке для согласования некоторого набора параметров и общих значений. Ее результатом становится так называемая *общая*, или *главная, ссылочная строка* (common reference string, CRS). Однако в случае с ZKP настройки могут быть гораздо более ограничивающими и даже опасными, чем кажется на первый взгляд. Существует три варианта настройки таких схем:
  - *доверенная.* Это означает, что участник, создавший CRS, имеет доступ и к секретам, что позволяет ему подделывать доказательства (поэтому такие секреты иногда называют токсичными отходами). Это довольно проблематично, поскольку мы снова имеем доверенную третью сторону, однако схемы, обладающие этим свойством, часто оказываются наиболее эффективными и предусматривают наименьший размер доказательства. Чтобы снизить риск, можно использовать MPC (многосторонние вычисления), распределив создание этих опасных параметров между многими участниками. Если хотя бы один участник окажется честным и удалит свои ключи после церемонии, это позволит избавиться от токсичных отходов;
  - *универсальная.* Доверенная схема считается универсальной, если с ее помощью можно доказать выполнение любой схемы, ограниченной некоторым размером. В противном случае она будет специфичной для какой-то одной схемы;
  - *прозрачная.* К счастью, многие схемы предусматривают и прозрачную версию, которая не требует участия доверенной третьей стороны в создании параметров системы. Прозрачные схемы универсальны по своей сути.

- *Квантовая устойчивость.* Некоторые ZKP-схемы используют криптографию с открытым ключом и продвинутые примитивы вроде билинейных спариваний (о них мы поговорим позже), в то время как другие полагаются только на симметричную криптографию, например хеш-функции, что делает их квантово-устойчивыми (правда, эта устойчивость обычно достигается за счет значительного увеличения размера доказательств).

Поскольку сейчас схемы zk-SNARK приобретают все более широкое распространение, далее я расскажу о том, как они работают.

15.3.1. Принцип работы схем zk-SNARK

Прежде всего следует сказать, что существует огромное количество схем zk-SNARK. Большинство из них состоит из следующих компонентов:

- системы доказательств, позволяющей доказывающему убедить в чем-то проверяющего;
- перевода или компиляции программы в нечто такое, что система доказательств может доказать.

С первым из них все вполне понятно, а вот второй требует пояснений. Для начала разберемся с первым компонентом.

Основная идея zk-SNARK заключается в доказательстве того, что вам известен некий многочлен $f(x)$, имеющий некоторые корни. Под корнями я подразумеваю то, что проверяющий имеет в виду некоторые значения (например, 1 и 2), а доказывающий должен доказать, что при этих значениях секретный многочлен обращается в 0 (например, $f(1) = f(2) = 0$). Кстати, многочлен, корнями которого являются значения 1 и 2, как в нашем примере, можно записать в виде $f(x) = (x - 1)(x - 2)h(x)$ для некоторого многочлена $h(x)$. (Если сомневаетесь, попробуйте вычислить значение этого выражения при $x = 1$ и $x = 2$.) Мы говорим, что доказывающий должен доказать, что он знает $f(x)$ и $h(x)$ такие, что $f(x) = t(x)h(x)$ для некоторого целевого многочлена $t(x) = (x - 1)(x - 2)$. В данном примере корнями, которые хочет уточнить проверяющий, являются значения 1 и 2.

И на этом все! Системы доказательства zk-SNARK обычно представляют лишь то, что доказывает, что вам известен некоторый многочлен. Я повторяю это, потому что, когда я впервые об этом узнал, для меня это не имело никакого смысла. Как можно доказать, что вы знаете некие секретные входные данные для программы, если можете доказать лишь то, что вам известен некий многочлен? Вот почему вторая часть схемы zk-SNARK, связанная с преобразованием программы в многочлен, так сложна. Но об этом чуть позже.

Вернемся к нашей системе доказательств: как доказать, что вам известна некая функция $f(x)$? Для этого вы должны доказать, что знаете функцию $h(x)$ такую, которая позволяет записать $f(x)$ в виде $f(x) = t(x)h(x)$. Ой... не так быстро. Мы же говорим о доказательствах *с нулевым разглашением*, верно? Как же можно доказать это, не выдавая $f(x)$? Для этого можно использовать следующие три трюка:

- *гомоморфные обязательства* — схему обязательств, подобную той, которую мы задействовали в других ZKP (о которых говорили в главе 7);

- *билинейные спаривания* — математическую конструкцию, обладающую некоторыми интересными свойствами, подробнее о которых поговорим позже;
- тот факт, что *разные многочлены в большинстве случаев принимают разные значения*.

Разберемся с каждым из этих трюков.

15.3.2. Гомоморфные обязательства для сокрытия частей доказательства

Первый трюк заключается в использовании *обязательств* для сокрытия значений, которые отправляем доказывающему. Однако, помимо этого, мы хотим позволить проверяющему выполнять над ними некоторые операции, чтобы он мог проверить доказательство. В частности, он должен проверить, что, если доказывающий принимает на себя обязательство, фиксируя значение многочлена $f(x)$, а также $h(x)$, мы имеем:

$$com(f(x)) = com(t(x))com(h(x)) = com(t(x)h(x)),$$

где обязательство $com(t(x))$ вычисляется проверяющим в виде согласованного ограничения, накладываемого на многочлен. Эти операции называются *гомоморфными*, и мы не смогли бы их выполнить, если бы в качестве механизмов принятия обязательств использовались хеш-функции (как показано в главе 2). Благодаря гомоморфным обязательствам мы можем «спрятать значения в экспоненте» (например, для фиксации значения v отправить обязательство $g^v \bmod p$) и выполнить следующие проверки тождества:

- равенство обязательств. Равенство $g^a = g^b$ означает, что $a = b$;
- сложение обязательств. Равенство $g^a = g^b g^c$ означает, что $a = b + c$;
- возведение обязательств в степень. Равенство $g^a = (g^b)^c$ означает, что $a = bc$.

Обратите внимание: последняя проверка работает только в том случае, если c является открытым значением, а не обязательством (g^c). С помощью одних только гомоморфных обязательств мы не можем проверить перемножение обязательств, а нам нужно именно это. К счастью, существует криптографический инструмент, позволяющий получить эти скрытые в экспонентах уравнения. Он называется *билинейными спариваниями*.

15.3.3. Билинейные спаривания для улучшения гомоморфных обязательств

Мы используем билинейные спаривания в схеме zk-SNARK *только для того*, чтобы иметь возможность перемножать значения внутри обязательств. Я не хочу слишком углубляться в объяснение концепции билинейных спариваний, поэтому скажу лишь, что это еще один инструмент в нашем наборе, позволяющий перемножать элементы, которые раньше не могли быть перемножены, перемещая их из одной группы в другую.

Обозначив билинейное спаривание буквой e, как обычно и делается, мы получаем $e(g_1, g_2) = h_3$, где g_1, g_2 и h_3 — это генераторы для разных групп. Используем здесь один и тот же генератор слева ($g_1 = g_2$), что сделает спаривание симметричным. Мы можем применить билинейное спаривание для выполнения операций умножения, скрытых в экспоненте, с помощью следующего уравнения:

$$e(g^b, g^c) = e(g)^{bc}.$$

И вновь используем билинейные спаривания для того, чтобы сделать наши обязательства гомоморфными не только относительно сложения, но и относительно умножения. (Обратите внимание на то, что эта схема не полностью гомоморфна, так как допускает только одну операцию умножения.) Билинейные спаривания постепенно получают все более широкое распространение в сфере криптографии. Они уже применяются в схемах гомоморфного шифрования, а также цифровой подписи, таких как BLS (о ней я упоминал в главе 8).

15.3.4. Откуда берется лаконичность

Наконец, *лаконичность* zk-SNARK обусловлена тем, что в большинстве случаев две разные функции принимают различные значения в разных точках. Что это значит для нас? Допустим, мне неизвестен многочлен $f(x)$, имеющий корни, которые мы выбрали вместе с проверяющим. Это означает, что $f(x) \neq t(x)h(x)$. Тогда результат вычисления $f(x)$ и $t(x)h(x)$ в случайной точке r не будет одинаковым *в большинстве случаев*. Почти для всех r получится $f(r) \neq t(r)h(r)$. В этом заключается суть так называемой *леммы Шварца — Зиппеля* (рис. 15.12).



Рис. 15.12. Лемма Шварца — Зиппеля гласит, что графики двух разных многочленов n-й степени могут пересекаться не более чем в n точках. Другими словами, в большинстве точек два разных многочлена будут иметь разные значения

Если мы знаем это, то нам достаточно доказать, что $com(f(r)) = com(t(r)h(r))$ для некоторой случайной точки r. Вот чем объясняется столь малый размер

доказательств zk-SNARK: сравнивая точки в группе, вы в итоге сравниваете гораздо бóльшие многочлены. По этой же причине большинство схем zk-SNARK требует доверенной настройки. Если доказывающий знает, какая случайная точка r будет использоваться для проверки равенства, то он может подделать недействительный многочлен, который все равно позволит подтвердить это равенство. Таким образом, доверенная настройка предполагает выполнение следующих действий.

1. Создание случайного значения r.
2. Фиксацию в виде обязательства различных результатов возведения значения r в степень (например, $g, g^r, g^{r^2}, g^{r^3}...$), чтобы доказывающий мог использовать эти значения для вычисления своего многочлена, не зная значения точки r.
3. Уничтожение значения r.

Понятен ли вам смысл пункта 2? Если моим, как доказывающего, многочленом является $f(x) = 3x^2 + x + 2$, то достаточно вычислить $(g^{r^2})^3 g^r g^2$, чтобы получить обязательство, фиксирующее значение многочлена в случайной неизвестной мне точке r.

15.3.5. От программ к многочленам

До сих пор единственное ограничение, налагаемое на многочлен, заключалось в том, что он должен иметь корни — некоторые значения, при которых обращается в 0 (что доказывающий должен доказать). Но как мы можем преобразовать более общее утверждение в доказательство знания многочлена? В криптовалютной сфере, в которой zk-SNARK в настоящее время используются наиболее активно, типичные утверждения формулируются следующим образом.

- Докажите, что значение находится в диапазоне $[0, 2^{64}]$ (это называется доказательством диапазона).
- Докажите, что (секретное) значение содержится в некотором заданном (публичном) дереве Меркла.
- Докажите, что сумма некоторых величин равна сумме некоторых других величин, и т. д.

Здесь кроется самая большая сложность. Как я уже говорил, преобразовать программу в многочлен очень трудно. Хорошая новость заключается в том, что я не собираюсь рассказывать вам обо всех деталях, однако сообщу достаточно для того, чтобы дать представление о том, как это работает. Этого хватит, чтобы вы могли понять, чего не хватает в моем объяснении, и заполнить пробелы самостоятельно. Далее мы сделаем следующее.

1. Сначала программа будет преобразована в арифметическую схему, подобную тем, что мы видели в разделе, посвященном MPC.
2. Эта арифметическая схема будет преобразована в систему уравнений определенной формы, называемую системой ограничений ранга 1, или R1CS.
3. Затем используем трюк, позволяющий преобразовать систему уравнений в многочлен.

15.3.6. Программы предназначены для компьютеров, а нам нужны арифметические схемы

Для начала предположим, что почти любую программу можно более или менее легко переписать на языке математики. Причина, по которой можно захотеть сделать это, очевидна: мы не можем доказать фрагменты кода, но можем доказать математические утверждения. Например, в листинге 15.1 представлена функция, в которой каждый вход является открытым, за исключением секретного входа a.

Листинг 15.1. Простая функция

```
fn my_function(w, a, b) {
    if w == true {
        return a * (b + 3);
    } else {
        return a + b;
    }
}
```

В данном случае, если все входы и выходы, кроме a, открытые, мы все равно можем найти значение a. Этот листинг также служит примером того, что не следует пытаться доказывать в режиме нулевого разглашения. Так или иначе, программу можно переписать на языке математики с помощью следующего уравнения:

$$w(a(b + 3)) + (1 - w)(a + b) = v,$$

где v — выход, а w равно либо 0 (ложь), либо 1 (истина). Обратите внимание на то, что это уравнение на самом деле не является ни программой, ни схемой. Скорее, оно напоминает ограничение. Если вы правильно выполните программу, а затем подставите в уравнение полученные входы и выходы, то равенство должно оказаться верным. Если оно неверно, значит, ваши входы и выходы не соответствуют правильному выполнению программы.

Вот как следует размышлять об этих многоцелевых ZKP. Вместо того чтобы выполнять функцию в режиме нулевого разглашения (что не имеет особого смысла), мы используем zk-SNARK, чтобы доказать: некоторые заданные входы и выходы соответствуют правильному выполнению программы, даже если часть из них опущена.

15.3.7. От арифметической схемы к системе ограничений ранга 1

Пока мы находимся лишь на первом этапе преобразования своей программы в нечто такое, что можно доказать с помощью zk-SNARK. Следующий этап предполагает ее преобразование в серию ограничений, которые затем можно преобразовать в доказательство знания некоторого многочлена. Для использования zk-SNARK нам необходима так называемая *система ограничений ранга 1* (rank-1 constraint system, R1CS), которая, по сути, представляет собой просто серию уравнений

вида $LR = O$, где L, R и O могут означать только сложение некоторых переменных. Таким образом, единственной операцией умножения является перемножение L и R. На самом деле причиной для преобразования арифметической схемы в такую систему уравнений является то, что это помогает получить итоговый результат, который мы можем доказать. Если вы попробуете сделать это, используя имеющееся уравнение, то получите что-то вроде следующего:

$$a(b + 3) = m;$$

$$w(m - a - b) = v - a - b.$$

На самом деле мы забыли про ограничение, согласно которому w может быть равно либо 0, либо 1. Но можем включить его в систему с помощью хитрого трюка:

$$a(b + 3) = m;$$

$$w(m - a - b) = v - a - b;$$

$$ww = w.$$

Чтобы понять смысл, попытайтесь взглянуть на эту систему как на набор ограничений: если вы даете мне несколько значений, которые, как вы утверждаете, соответствуют входам и выходам моей программы, то я должен быть в состоянии проверить верность равенств при этих значениях. Если одно из равенств неверно, то это означает, что предоставленные вами входы и выходы программы не соответствуют друг другу. Иными словами, zk-SNARK позволяет удалять входы или выходы из транскрипта правильного выполнения программы на основании проверки.

15.3.8. От системы ограничений ранга 1 к многочлену

Остается вопрос: как преобразовать эту систему в многочлен? Как всегда, для этого нам потребуется несколько трюков! Поскольку система состоит из трех различных уравнений, первым шагом будет согласование трех корней многочлена. Мы можем просто выбрать в качестве корней значения 1, 2, 3, которые будут означать, что многочлен $f(x) = 0$ при $x = 1$, $x = 2$ и $x = 3$. Так мы можем заставить многочлен представлять все уравнения системы одновременно: первое уравнение при вычислении значения многочлена в точке 1, второе уравнение — при вычислении значения в точке 2 и т. д. Теперь задача доказывающего состоит в создании многочлена $f(x)$ такого, что:

$$f(1) = a(b + 3) - m;$$

$$f(2) = w(m - a - b) - (v - a - b);$$

$$f(3) = ww - w.$$

Обратите внимание на то, что все эти уравнения должны обращаться в 0, если значения точно соответствуют выполнению исходной программы. Другими словами, многочлен $f(x)$ имеет корни 1, 2, 3 только в том случае, если мы создали его правильно. В этом и заключается суть zk-SNARK: у нас есть протокол для

доказательства того, что многочлен $f(x)$ действительно имеет эти корни, известные как доказывающему, так и проверяющему.

Однако на этом объяснение не заканчивается, потому что теперь проблема заключается в том, что доказывающий имеет слишком большую свободу при выборе многочлена $f(x)$. Он может просто найти многочлен с корнями 1, 2, 3, не заботясь о значениях a, b, m, v и w. Он может делать практически все, что захочет! Поэтому нужна система, позволяющая зафиксировать все части многочлена, кроме секретных значений, которые *не* должны быть известны проверяющему.

15.3.9. Для вычисления значения многочлена, скрытого в экспоненте, нужны двое

Итак, нам нужен доказывающий, который должен правильно выполнить программу с использованием своего секретного значения a и открытых значений b и w и получить результат v, который может опубликовать. Затем он должен создать многочлен, заполнив только те части, которые не должны быть известны проверяющему, то есть значения a и m. Таким образом, нам нужно, чтобы в рамках реального протокола zk-SNARK доказывающий имел как можно меньше свободы при создании своего многочлена и вычислении его значения в случайной точке.

Для этого многочлен должен создаваться динамически — так, чтобы доказывающий заполнял только свои части, а проверяющий — все остальные. Например, возьмем первое уравнение, $f(1) = a(b + 3) - m$, и представим его в виде:

$$f_1(x) = aL_1(x)(b + 3)R_1(x) - mO_1(x),$$

где $L_1(x)$, $R_1(x)$, $O_1(x)$ — это многочлены, которые имеют значение 1 при $x = 1$ и 0 при $x = 2$ и $x = 3$. Это необходимо для того, чтобы они влияли только на первое уравнение. (Такие многочлены довольно легко найти с помощью алгоритмов вроде метода интерполяции Лагранжа.) Теперь обратите внимание еще на две вещи.

- В качестве коэффициентов многочленов выступают входы, промежуточные значения и выходы.
- Многочлен $f(x)$ — это сумма $f_1(x) + f_2(x) + f_3(x)$, где $f_2(x)$ и $f_3(x)$ представляют уравнения 2 и 3 аналогично тому, как было в случае с $f_1(x)$.

Как видите, первое уравнение по-прежнему представлено в точке $x = 1$:

$$f(1) = f_1(1) + f_2(1) + f_3(1) = f_1(1) =$$
$$= aL_1(1)(b + 3)R_1(1) - mO_1(1) = a(b + 3) - m.$$

Благодаря этому новому способу представления уравнений (которые, как вы помните, отражают процесс выполнения программы) доказывающий может найти значения интересующих его частей многочлена следующим образом.

1. Возведением в степень случайной точки r, скрытой в экспоненте, для восстановления многочленов $L^1(r)$ и $O^1(r)$.

2. Возведением $g^{L_1(r)}$ в степень с использованием секретного значения a для получения $(g^{L_1(r)})^a = g^{aL_1(r)}$, что представляет собой скрытое в экспоненте значение $aL^1(x)$ в неизвестной и случайной точке $x = r$.
3. Возведением $g^{O_1(r)}$ в степень с помощью секретного промежуточного значения m для получения $(g^{O_1(r)})^m = g^{mO_1(r)}$, что представляет собой скрытое в экспоненте значение $mO^1(x)$ в случайной точке r.

Затем проверяющий может заполнить недостающие части, восстановив $(g^{R_1(r)})^b$ и $(g^{R_1(r)})^3$ для некоторого согласованного значения b с помощью тех же методов, которые использовал доказывающий. Сложив их, проверяющий получает сумму $g^{bR_1(r)} + g^{3R_1(r)}$, которая представляет собой скрытое значение $(b + 3)R_1(x)$ в неизвестной и случайной точке $x = r$. Наконец, проверяющий может восстановить скрытую в экспоненте часть $f_1(r)$ с помощью билинейного спаривания:

$$e(g^{aL_1(r)}, g^{(b+3)R_1(r)}) - e(g, g^{mO_1(r)}) = e(g, g)^{aL_1(r)}\,(b + 3)R_1(r) - mO_1(r).$$

Экстраполировав эти методы на весь многочлен $f(x)$, можно выяснить все детали итогового протокола. Разумеется, это грубое упрощение реальной схемы zk-SNARK, так как по-прежнему оставляет доказывающему слишком большую свободу.

Все остальные трюки, используемые в протоколах zk-SNARK, призваны еще сильнее ограничить возможности доказывающего, чтобы гарантировать правильное и последовательное заполнение им недостающих частей, а также оптимизировать все то, что может быть оптимизировано. Кстати, самое лучшее объяснение, которое мне попадалось, — это статья Максима Петкуса *Why and How zk-SNARK Works: Definitive Explanation*, в которой очень подробно объясняется все то, что я упустил из виду.

На этом закончим введение в тему zk-SNARK. На практике протокол zk-SNARK гораздо более сложен для понимания и применения! Мало того что объем работы по преобразованию программы в нечто такое, что можно доказать, оказывается весьма существенным, он иногда еще и накладывает дополнительные ограничения на криптографический протокол. Например, распространенные хеш-функции и схемы цифровой подписи часто оказываются слишком тяжеловесными для многоцелевых ZKP-схем, что побуждает многих разработчиков протоколов исследовать другие, дружественные к ZKP схемы. Кроме того, как я уже говорил, существует огромное количество схем, как использующих, так и не использующих zk-SNARK, которые могут оказаться более уместными в качестве многоцелевых ZKP-схем в той или иной ситуации.

К сожалению, не существует ZKP-схемы на все случаи жизни, например прозрачной, лаконичной, универсальной и квантово-устойчивой, и не всегда ясно, какую из них следует применять в тех или иных ситуациях. Эта область еще слишком молода и каждый год пополняется новыми и более эффективными схемами. Вполне возможно, что через несколько лет появятся более совершенные стандарты и простые в применении библиотеки, так что, если вам интересно это направление, следите за его развитием!

РЕЗЮМЕ

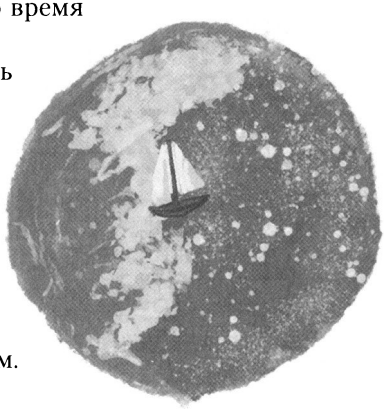
- За последнее десятилетие многие теоретические криптографические примитивы были значительно усовершенствованы в плане эффективности и практичности, некоторые из них уже применяются в реальном мире.
- Защищенные многосторонние вычисления (MPC) — это примитив, который позволяет нескольким участникам правильно выполнить программу, не раскрывая при этом свои входные данные. Пороговые подписи начинают применяться в криптовалютной сфере, а пересечения закрытых множеств (PSI) используются в современных масштабных протоколах, таких как функция Google Password Checkup.
- Полностью гомоморфное шифрование (FHE) позволяет вычислять произвольные функции на основе зашифрованных данных без их расшифровки. Оно может найти применение в облачных сервисах, предоставляя доступ к данным только их владельцу, но при этом позволяя облачной платформе выполнять над этими данными полезные вычислительные операции.
- Многоцелевые доказательства с нулевым разглашением (ZKP) уже нашли множество применений благодаря появившемуся недавно способу создания небольших и быстро проверяемых доказательств. В основном они используются в криптовалютной сфере для повышения конфиденциальности или уменьшения размера блокчейна. Однако их потенциал этим не исчерпывается, и по мере появления более совершенных стандартов и более простых в применении библиотек область их применения, скорее всего, будет расширяться.

16 Условия, в которых криптография может не сработать

В этой главе

- ✓ Распространенные проблемы, с которыми можно столкнуться при использовании криптографических инструментов.
- ✓ Принципы, которым нужно следовать, чтобы создать хорошую криптографическую систему.
- ✓ Обязанности специалиста по криптографии и подстерегающие его опасности.

Итак, вы проделали долгий путь. Вы достигли последней главы книги, но путешествие на этом не заканчивается. Теперь у вас есть все необходимое для того, чтобы войти в реальный мир криптографии. Пришло время применить полученные знания на практике.



Перед тем как расстаться с вами, я хотел бы дать несколько подсказок и инструментов, которые пригодятся вам в дальнейшем. Квесты, которые вам предстоит проходить, обычно развиваются по одной и той же схеме: все начинается с возникновения проблемы, которая побуждает отправиться на поиски существующего криптографического примитива или протокола. Далее вы пытаетесь отыскать стандарт и хорошую реализацию, а затем применить ее наилучшим образом. Это при условии, что все пойдет по плану...

ПРЕЖДЕ ЧЕМ МЫ РАССТАНЕМСЯ

Тому, кто стремится преодолеть разрыв между теорией и практикой, придется сразить множество драконов. Вот ваш меч — возьмите его.

16.1. ПОИСК ПОДХОДЯЩЕГО КРИПТОГРАФИЧЕСКОГО ПРИМИТИВА ИЛИ ПРОТОКОЛА — СКУЧНОЕ ЗАНЯТИЕ

Представьте, что вам нужно зашифровать трафик, обеспечить взаимную аутентификацию нескольких серверов или хранить секреты так, чтобы они не превратились в единую точку отказа. Что вы будете делать?

Для шифрования трафика можно использовать протокол TLS или Noise (см. главу 9). Для аутентификации серверов вы можете создать инфраструктуру открытых ключей (см. главу 9) и задействовать подпись центра сертификации. А во избежание компрометации всей системы в случае компрометации одного секрета — распределить этот секрет с помощью пороговой схемы (см. главу 8). Это вполне приемлемые ответы.

Если проблема, с которой вы столкнулись, типична, то вполне вероятно, что удастся найти существующий криптографический примитив или протокол, предназначенный для решения стоящей перед вами задачи. Эта книга призвана познакомить вас со стандартными примитивами и распространенными протоколами, так что к этому моменту вы должны иметь представление о том, какие инструменты можно использовать при столкновении с той или иной криптографической проблемой.

Криптография — это довольно интересная область, которая постоянно развивается, пополняясь все новыми примитивами. Хотя у вас может возникнуть соблазн применить экзотические методы криптографии для решения своей проблемы, старайтесь придерживаться консервативного подхода. Помните о том, что *сложность — это враг безопасности*. Когда вы что-то делаете, старайтесь выбирать самый простой путь. Слишком много уязвимостей появилось вследствие использования экстравагантных решений. В 2015 году Бернштейн назвал эту концепцию *скучной криптографией* (boring cryptography), подтолкнув компанию Google к тому, чтобы назвать свою TLS-библиотеку BoringSSL.

> Предложенные криптографические инструменты должны выдержать многолетнюю тщательную проверку, прежде чем стать вероятными кандидатами на использование в полевых условиях. Особенно если они основаны на новых математических задачах.
>
> *Ривест и др. (Responses to NIST's proposal, 1992)*

Но что, если вы не можете найти криптографический примитив или протокол, позволяющий решить вашу проблему? В этом случае придется окунуться в мир *теоретической* криптографии, которая, очевидно, не является темой этой книги. Однако я могу порекомендовать другие источники.

Первой рекомендацией является бесплатная книга *A Graduate Course in Applied Cryptography*, написанная Дэном Боне и Виктором Шоупом и доступная на сайте https://cryptobook.us. Она охватывает все, о чем вы прочитали в этой книге, но дает гораздо более глубокие объяснения. У Дэна Боне также есть замечательный бесплатный онлайн-курс *Cryptography I*, доступный по адресу https://www.coursera.org/learn/crypto и представляющий собой гораздо более мягкое введение в теоретическую криптографию. Если вам нужно что-то находящееся на полпути между этой книгой и миром теоретической криптографии, могу порекомендовать книгу Жана-Филиппа Омассона *Serious Cryptography: A Practical Introduction to Modern Encryption*[1].

Теперь представим, что у вас уже *есть* криптографический примитив или протокол, способный решить проблему. Этот криптографический примитив или протокол все еще остается в значительной степени теоретической вещью. Так что было бы здорово, если бы у него был практический стандарт, который вы могли бы использовать прямо сейчас.



16.2. КАК ИСПОЛЬЗОВАТЬ КРИПТОГРАФИЧЕСКИЙ ПРИМИТИВ ИЛИ ПРОТОКОЛ. ВЕЖЛИВЫЕ СТАНДАРТЫ И ФОРМАЛЬНАЯ ВЕРИФИКАЦИЯ

Итак, вы знаете, что решение, соответствующее вашим потребностям, существует, но есть ли у него стандарт? В отсутствие стандарта примитив часто предлагается в том виде, который не учитывает условия его реального применения. Криптографы обычно не задумываются о различных подводных камнях, связанных с применением их примитива или протокола, а также о деталях их реализации. Стандарты, которые заботятся о реализации и не оставляют пользователям возможности выстрелить себе в ногу, Риад С. Вахби назвал *вежливой криптографией*.

[1] *Омассон Ж.-Ф.* О криптографии всерьез. Практическое введение в современное шифрование. – 2021.

> Стандарт не должен предоставлять бедному пользователю достаточно веревки для того, чтобы повеситься.
>
> *Ривест и др. (Responses to NIST's proposal, 1992)*

Вежливый стандарт — это спецификация, призванная охватить все крайние случаи и потенциальные проблемы безопасности, предоставляя безопасные и простые в использовании интерфейсы, а также давая рекомендации относительно применения соответствующего примитива или протокола. Кроме того, хорошие стандарты предусматривают тестовые векторы — списки соответствующих друг другу входных и выходных данных, которые вы можете подать на вход своей реализации для проверки правильности ее работы.

К сожалению, не все стандарты вежливые, и создаваемые ими криптографические ловушки становятся причиной появления большинства уязвимостей, о которых я рассказал в этой книге. Иногда стандарты оказываются слишком расплывчатыми, не предусматривают тестовых векторов или пытаются одновременно сделать слишком много. Например, существует термин *«криптографическая гибкость»*, используемый для определения гибкости протокола в плане поддерживаемых им криптографических алгоритмов. Поддержка различных криптографических алгоритмов может дать стандарту преимущество, потому что иногда один алгоритм выходит из строя и устаревает, а другие — нет. В такой ситуации негибкий протокол может помешать клиентам и сервисам двигаться дальше. В то же время излишняя гибкость может сильно повлиять на сложность стандарта, а иногда даже привести к появлению уязвимостей, о чем свидетельствуют многочисленные атаки, направленные на понижение версии протокола TLS.

Очень часто проблемы возникают тогда, когда ваша задача представляет собой крайний случай, на который не рассчитаны распространенные примитивы или протоколы, либо когда она не соответствует стандартизированному решению. По этой причине разработчики часто создают собственные мини-протоколы или мини-стандарты. Вот тут-то и начинаются проблемы.

Неверные предположения относительно модели угроз примитива (того, от чего он защищает) или его компонуемости (способа его применения в рамках протокола) приводят к поломкам. Эти контекстно-специфические проблемы усугубляются тем, что в процессе создания криптографических примитивов разработчик обычно не продумывает все проблемы, которые могут возникнуть при использовании примитива тем или иным способом или в рамках другого примитива или протокола. Примеры — алгоритм X25519, ломающийся в крайних случаях (см. главу 11), подписи, считающиеся уникальными (см. главу 7), и двусмысленность, возникающая при подписании сообщения (см. главу 10). И это не обязательно происходит по вашей вине! Разработчики превзошли криптографов, обнаружив подводные камни, о существовании которых никто не подозревал.

Оказавшись в подобной ситуации, криптограф обычно берет ручку и бумагу и выстраивает доказательство. Однако для нас, практиков, этот подход не очень удобен, потому что у нас либо нет времени на такую работу (а она действительно занимает много времени), либо нет нужного опыта. Тем не менее мы не беспомощны. Мы можем использовать компьютеры для облегчения анализа мини-протокола. Эта так называемая *формальная верификация* может стать прекрасным вложением вашего времени.

Формальная верификация позволяет вам написать протокол на промежуточном языке и проверить некоторые его свойства. Например, *Tamarin protocol prover* (рис. 16.1) — это инструмент формальной верификации, который используется для выявления незаметных атак на различные протоколы. Подробнее об этом можно узнать, ознакомившись со статьями *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman* (2019) и *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures* (2019).

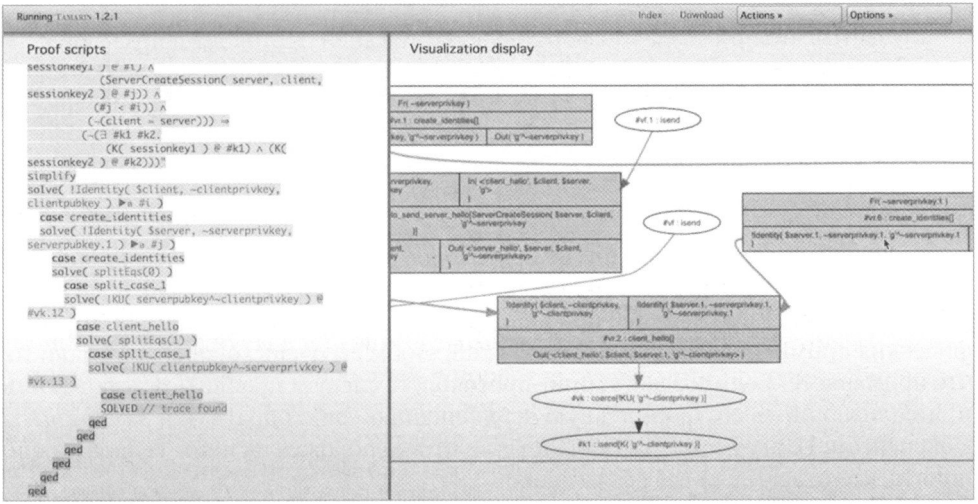


Рис. 16.1. Tamarin protocol prover — это бесплатный инструмент формальной верификации, который можно применять для моделирования криптографического протокола и выявления атак, которым он подвергается

Обратная сторона медали заключается в том, что работать с инструментами формальной верификации часто бывает трудно. Сначала необходимо найти способ перевода протокола и используемых инструментом концепций на некий формальный язык, что зачастую бывает не так уж просто. А после описания протокола на формальном языке вам еще нужно понять, что именно вы хотите доказать и как это можно выразить на этом языке.

Нередко можно увидеть доказательство, которое доказывает совсем не то, что нужно, в связи с чем возникает вопрос о том, кто проверяет результаты формальной верификации. Некоторые перспективные исследования в этой области направлены на то, чтобы облегчить разработчикам задачу проведения формальной верификации протоколов. Например, инструмент Verifpal (https://verifpal.com) обеспечивает компромисс между надежностью (возможностью выявления всех атак) и простотой применения.

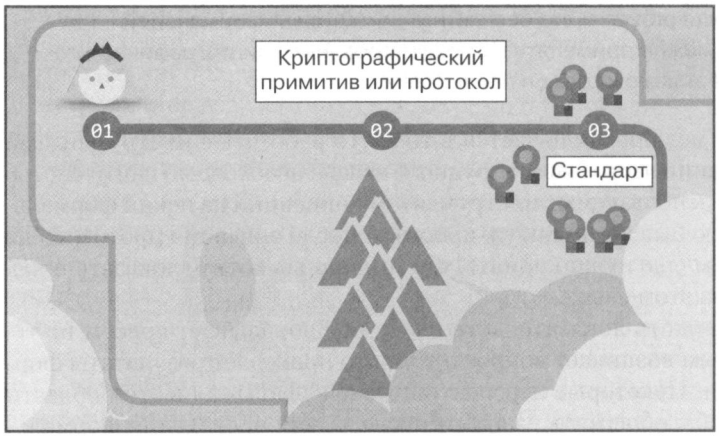
Вы также можете использовать формальную верификацию для проверки доказательств безопасности криптографического примитива с помощью таких инструментов, как Coq, CryptoVerif и ProVerif, и даже генерирования формально верифицированных реализаций на разных языках (см. проекты HACL*, Vale и fiat-crypto, которые реализуют основные криптографические примитивы с такими

проверенными свойствами, как корректность, безопасность доступа к памяти и т. д.). Тем не менее формальная верификация не является абсолютно надежным методом: разрывы между протоколом, изложенным на бумаге, и его формальным описанием или между формальным описанием и реализацией всегда будут существовать и казаться безобидными до тех пор, пока не приведут к фатальным последствиям.

АТАКА KRACK

Случается, что при создании формального описания протокола допускаются критические расхождения с его фактической реализацией, что впоследствии приводит к появлению брешей и реальным атакам. Именно это произошло в 2017 году, когда в ходе атаки KRACK (https://krackattacks.com) был взломан протокол Wi-Fi WPA2, который до этого был формально верифицирован.

Изучение причин провала других протоколов помогает избежать тех же ошибок. Конкурсы, проводимые на платформах cryptopals.com или cryptohack.org, позволяют узнать, что может пойти не так при использовании и комбинировании криптографических примитивов и протоколов. Важно, чтобы вы очень хорошо понимали то, что применяете! Если создаете мини-протокол, то следует проявить осторожность и либо выполнить его формальную верификацию, либо обратиться за помощью к экспертам. Итак, у нас есть стандарт или что-то похожее на него. Теперь нужно понять, кто отвечает за его реализацию.



16.3. ГДЕ ИСКАТЬ ХОРОШИЕ БИБЛИОТЕКИ

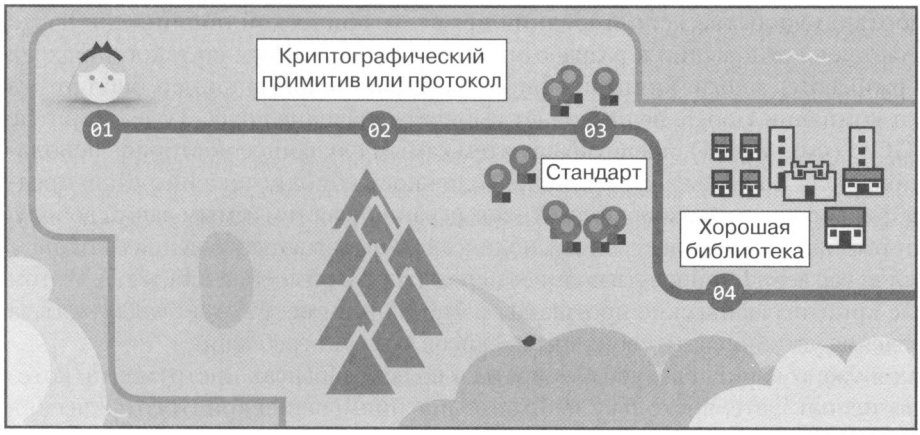
Вы на шаг приблизились к решению своей проблемы. Вы знаете, какой примитив или протокол хотите использовать, и у вас есть соответствующий стандарт. В то же время вы на шаг отдалились от спецификации, а это значит, что можете допустить ошибки. Однако сначала нужно понять, откуда брать код.

Оглянувшись вокруг, вы видите множество библиотек или фреймворков, которые можно применять. Но какую именно библиотеку следует выбрать? Какая из них наиболее безопасна? На этот вопрос трудно ответить. Некоторые библиотеки пользуются заслуженным авторитетом, и я упомянул несколько в книге, в частности Tink от компании Google, libsodium, cryptography.io и др.

Однако иногда найти хорошую библиотеку бывает трудно. Возможно, используемый вами язык программирования не обеспечивает достаточно обширную поддержку криптографии, а возможно, выбранный примитив или протокол не предусматривает достаточного количества реализаций. В таких ситуациях следует проявить осторожность и попросить совета у представителей криптографического сообщества, обратиться к авторам библиотеки и, возможно, даже попросить экспертов сделать обзор кода. Например, сообщество r/crypto на платформе Reddit довольно отзывчивое и доброжелательное. Иногда срабатывает прямая переписка с авторами, вопросы аудитории во время открытых сессий на конференциях тоже могут принести свои плоды.

Если же вы оказались в безвыходной ситуации, то, возможно, придется самостоятельно реализовать криптографический примитив или протокол. При этом вы можете столкнуться со множеством проблем, так что имеет смысл узнать о трудностях, часто возникающих в процессе реализации криптографических инструментов. К счастью, следование хорошему стандарту снижает вероятность допустить ошибку. И все же создание криптографических инструментов — это искусство, за которое не стоит браться без особой необходимости.

Один из интересных способов тестирования криптографической реализации — использование специальных *инструментариев*. Хотя ни один инструментарий не способен охватить все криптографические алгоритмы, Wycheproof компании Google заслуживает упоминания. Он представляет собой набор тестовых векторов, которые можно применять для поиска ошибок в таких распространенных криптографических алгоритмах, как ECDSA, AES-GCM и т. д. С помощью этого фреймворка было выявлено впечатляющее количество ошибок в различных криптографических реализациях. Теперь представим, что вы решили реализовать криптографический алгоритм не самостоятельно, а с помощью библиотеки.



16.4. ЗЛОУПОТРЕБЛЕНИЕ КРИПТОГРАФИЕЙ. РАЗРАБОТЧИКИ — ГЛАВНЫЕ ВРАГИ

Итак, вы нашли код, который можно использовать, и сделали еще один шаг вперед, но при этом обнаружили, что возможностей ошибиться стало еще больше. Именно на этом этапе допускают большинство ошибок в прикладной криптографии. В книге мы неоднократно сталкивались с примерами злоупотребления криптографией, в частности с проблемами, вызванными повторным использованием nonce в таких алгоритмах, как ECDSA (см. главу 7) и AES-GCM (см. главу 4), с коллизиями, возникающими при неправильном применении хеш-функций (см. главу 2), с возможностью выдать себя за кого-то другого при отсутствии аутентификации источника (см. главу 9) и т. д.

> Результаты показывают, что только 17 % ошибок, приводящих к катастрофическим последствиям, находятся в самих криптографических библиотеках, а остальные 83 % связаны с неправильным использованием криптографических библиотек отдельными приложениями.
>
> *Дэвид Лазар, Хаоган Чен, Си Ванг и Николай Зельдович (Why does cryptographic software fail? A case study and open problems, 2014)*

В целом чем более абстрактным является примитив или протокол, тем безопаснее его применять. Например, сервис AWS предусматривает службу управления ключами KMS (Key Management Service), позволяющую размещать свои ключи в HSM и выполнять криптографические вычисления по требованию. Таким образом, криптография абстрагируется на уровне приложений. Другой пример — языки программирования, предоставляющие криптографические инструменты в рамках своих стандартных библиотек, которые зачастую оказываются более надежными по сравнению с библиотеками сторонних разработчиков. Одним из превосходных примеров является стандартная библиотека Golang.

Забота об удобстве использования криптографической библиотеки часто выражается в «отношении к разработчику как к врагу». Именно такого подхода придерживаются многие криптографические библиотеки. Например, библиотека Tink от компании Google не позволяет выбирать значение nonce/IV в алгоритме AES-GCM (см. главу 4), предотвращая тем самым случайное повторное использование nonce. Во избежание излишней сложности библиотека libsodium предлагает фиксированный набор примитивов, ограничивая тем самым вашу свободу. Некоторые библиотеки электронных подписей оборачивают сообщения в подпись, заставляя вас верифицировать подпись перед отправкой сообщения, и т. д. В этом смысле криптографические протоколы и библиотеки несут ответственность за обеспечение устойчивости своих интерфейсов к злоупотреблениям.

Как я уже говорил, следует разобраться во всех нюансах инструмента, который вы используете, поскольку неправильное применение криптографических

примитивов или протоколов может привести к катастрофическим последствиям. Изучите стандарты и соображения безопасности, прочитайте руководства и документацию к криптографической библиотеке. Достаточно ли этого? Ну, не совсем, поскольку вы не единственный пользователь.



16.5. ВЫ ДЕЛАЕТЕ ЭТО НЕПРАВИЛЬНО: УДОБНАЯ БЕЗОПАСНОСТЬ

Чаще всего применение криптографии решает возникающие у приложений проблемы вполне прозрачными способами, но так бывает не всегда! Иногда в процесс использования криптографии вовлекаются пользователи приложений.

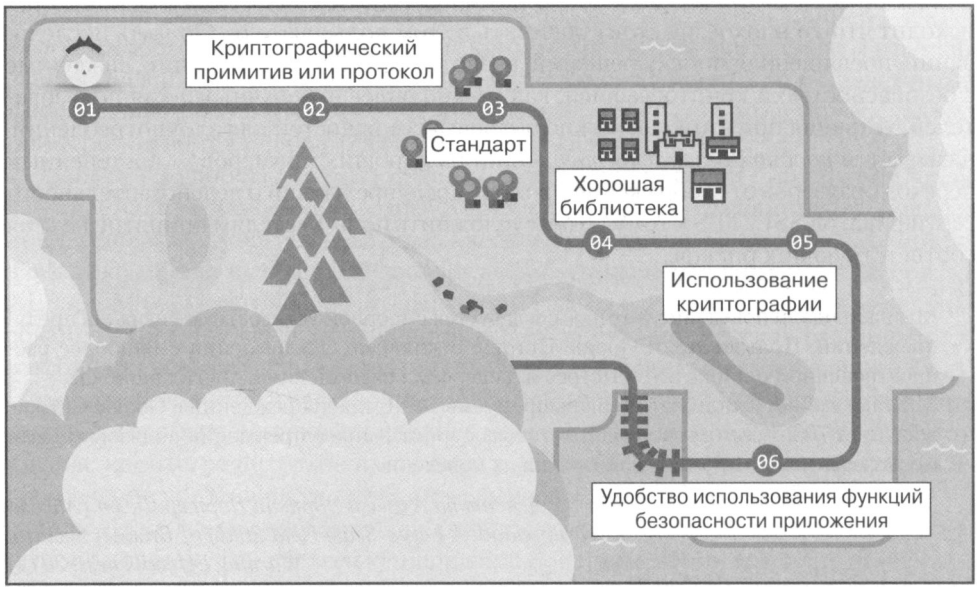
Как правило, обучение может помочь лишь незначительно. Поэтому, когда происходит что-то плохое, не стоит обвинять в этом пользователя. Область исследований, посвященная поиску решений, позволяющих сделать функции, связанные с безопасностью и криптографией, как можно более прозрачными для пользователей, устраняя при этом как можно больше возможностей для злоупотреблений, называется *удобной безопасностью*. Один из хороших примеров — постепенный переход браузеров от отображения простых предупреждений о недействительности сертификатов SSL/TLS к тому, чтобы усложнить пользователям принятие на себя соответствующих рисков.

> Мы наблюдали поведение, которое согласуется с теорией усталости от сигналов предупреждений. Пользователи Google Chrome отключают уведомления о наиболее распространенной ошибке SSL быстрее и чаще, чем уведомления о других ошибках. <...> Мы также обнаружили, что доля игнорируемых SSL-предупреждений в Google Chrome достигает 70,2 %, а это говорит о том, что восприятие предупреждений пользователями может оказывать огромное влияние на их поведение.
>
> *Девдатта Ахаве и Эдриенн Портер Фелт (Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness, 2013)*

Еще один хороший пример – переход от применения паролей к поддержке двухфакторной аутентификации (см. главу 11) в службах, чувствительных к безопасности. Поскольку заставить пользователей придумывать надежные и уникальные для каждого сервиса пароли слишком сложно, было найдено другое решение, устраняющее риск, связанный с компрометацией паролей. Сквозное шифрование также является хорошим примером, поскольку пользователи с трудом понимают, что означает сквозное шифрование их коммуникаций и как сильно безопасность зависит от выполняемой ими проверки отпечатков (см. главу 10). Всякий раз, когда применение криптографии передается на откуп пользователям, необходимо приложить немало усилий для того, чтобы снизить для них риск допустить ошибку.

ВРЕМЯ ДЛЯ РАССКАЗА

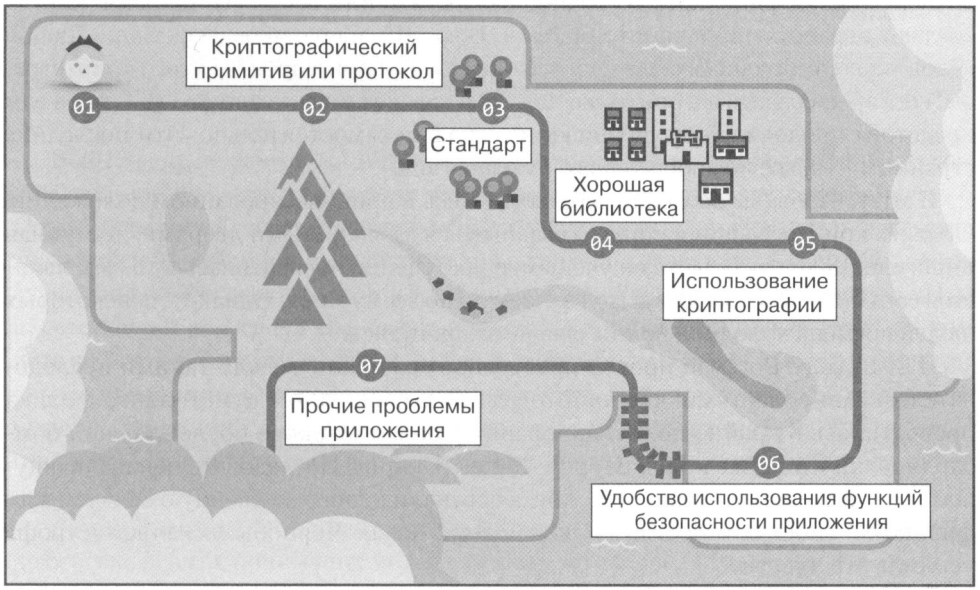
Много лет назад меня попросили проанализировать схему сквозного шифрования коммуникаций, предлагаемую широко распространенным приложением для обмена сообщениями. Она предполагала применение современного протокола Signal (см. главу 10), но не позволяла пользователям проверять открытые (или сеансовые) ключи других пользователей. Это означало, что, хотя ваши коммуникации подвергались сквозному шифрованию в присутствии пассивного противника, злонамеренный сотрудник мог легко обновить открытый ключ какого-нибудь пользователя или чьи-то сеансовые ключи, после чего вы не смогли бы обнаружить атаку посредника.



16.6. КРИПТОГРАФИЯ — ЭТО НЕ ИЗОЛИРОВАННЫЙ ОСТРОВ

Криптография часто выступает в качестве компонента более сложной системы, которая также может содержать ошибки. И большинство этих ошибок кроется в тех частях, которые не имеют отношения к самой криптографии. Как правило, злоумышленник ищет самое слабое звено цепи, самый низко висящий плод, а криптография часто вполне успешно справляется с поднятием планки. Поэтому в больших и сложных системах могут появляться более доступные векторы атак. Как сказал Ади Шамир, «криптографию обычно обходят, а не взламывают».

Поэтому, убедившись в том, что криптография в системе в достаточной степени консервативна, хорошо реализована и протестирована, имеет смысл столь же тщательно проверить остальные компоненты системы. В противном случае все ваши усилия могут оказаться напрасными.



16.7. ВАША ОБЯЗАННОСТЬ КАК ПРАКТИКУЮЩЕГО КРИПТОГРАФА СОСТОИТ В ТОМ, ЧТОБЫ НЕ РАЗВЕРТЫВАТЬ СОБСТВЕННУЮ КРИПТОГРАФИЮ

Итак, вы достигли конца книги и готовы отправиться в свободное плавание. Однако я должен предупредить вас о том, что книга написана не для того, чтобы наделить вас сверхспособностями, а чтобы передать вам ощущение хрупкости и понимание того, что криптографией можно легко злоупотребить и простейшая ошибка может привести к разрушительным последствиям. Будьте осторожны!

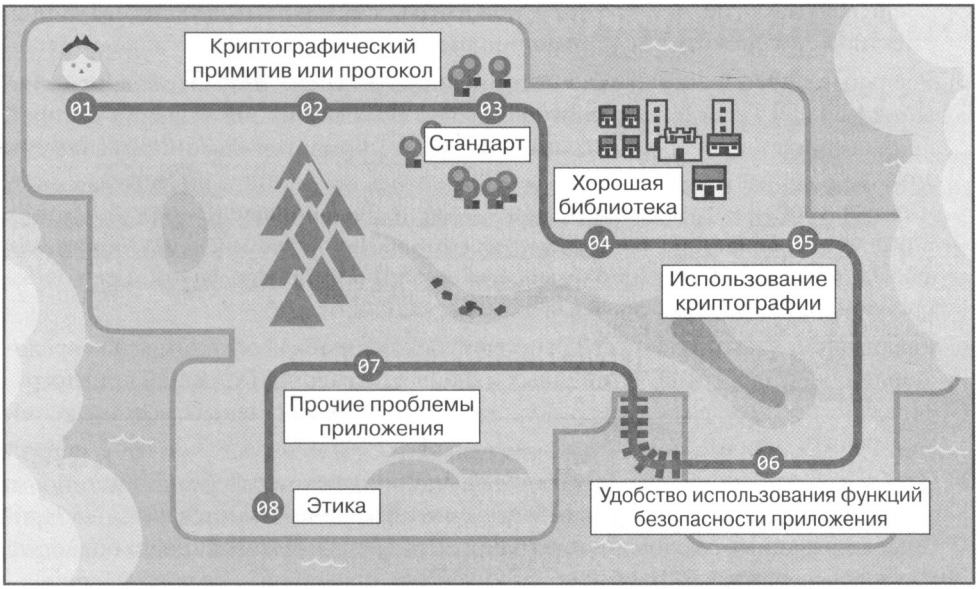
Теперь у вас на вооружении есть большой набор криптоинструментов. Вы уже должны уметь распознавать используемый тип криптографии и даже выявлять то, что кажется подозрительным. Должны уметь принимать проектные решения, применять криптографические инструменты в своем приложении и понимать, когда вы сами или кто-то другой начинает делать нечто опасное, что может потребовать более пристального внимания. Никогда не стесняйтесь спрашивать мнение эксперта.

«Не развертывайте собственную криптографию» — эту фразу по поводу криптографии, наверное, чаще всего можно услышать от инженеров-программистов. И они в чем-то правы. Хотя вы должны быть способны реализовывать или даже создавать собственные криптографические примитивы и протоколы, не следует использовать их в производственной среде. Создание эффективных криптографических инструментов требует многих лет изучения разнообразных тонкостей этой области, причем касающихся не только дизайна, но и криптоанализа. Даже эксперты, изучавшие криптографию на протяжении всей своей жизни, иногда создают неработающие криптосистемы. Брюс Шнайер однажды сказал: «Любой человек, от самого невежественного любителя до самого продвинутого криптографа, способен создать алгоритм, который он сам не сможет взломать». Теперь вам предстоит продолжить изучение криптографии самостоятельно. Эти последние страницы — вовсе не конец вашего путешествия.

Я хочу, чтобы вы осознали, что находитесь в привилегированном положении. Область криптографии начинала развиваться за закрытыми дверями, доступная лишь членам правительства и ученым, и постепенно превратилась в то, чем является сегодня, — в науку, открыто изучаемую по всему миру. Однако для некоторых людей времена холодной войны еще не закончились.

В 2015 году Рогауэй провел интересное сравнение между такими исследовательскими областями, как криптография и физика. Он отметил, что физика превратилась в крайне политизированную область вскоре после атомной бомбардировки Японии в конце Второй мировой войны. Исследователи начали ощущать глубокую ответственность, когда осознали непосредственную связь между физикой и гибелью множества людей. Чуть позже Чернобыльская катастрофа усилила это чувство.

В то же время криптографы часто говорят о конфиденциальности так, как будто она относится совсем к другой области, что делает исследования в этой сфере весьма аполитичными. Тем не менее решения, которые принимаем мы с вами, могут оказать продолжительное воздействие на наше общество. В следующий раз, когда будете разрабатывать или внедрять криптографическую систему, подумайте о своей модели угроз. Рассматриваете ли вы себя в качестве доверенной стороны или создаете систему так, чтобы даже сами не могли получить доступ к данным своих пользователей или повлиять на их безопасность? Как вы расширяете возможности пользователей с помощью криптографии? Что именно шифруете? Как сказал бывший глава АНБ Майкл Хайден, «мы убиваем людей на основе метаданных» (http://mng.bz/PX19).



В 2012 году недалеко от Санта-Барбары сотни криптографов собрались в темном лекционном зале, чтобы послушать доклад Джонатана Зиттрейна *The End of Crypto* (https://www.youtube.com/watch?v=3ijjHZHNIbU), с которым он выступал на самой значимой конференции по криптографии в мире под названием Crypto. В ходе выступления он показал аудитории ролик с фрагментом эпизода телесериала «Игра престолов». В нем евнух Варис задает деснице короля Тириона загадку: «В комнате сидят три важных человека — король, священник и богач. Между ними стоит обычный наемник. Каждый велит наемнику убить двух других. Кто выживет, а кто умрет?» Тирион не задумываясь отвечает: «Зависит от наемника», на что евнух говорит: «Если это решает наемник, то почему мы делаем вид, будто вся власть принадлежит королям?»

Затем Джонатан остановил ролик и спросил, обратившись к зрителям: «Надеюсь, вы, ребята, понимаете, что наемники — это вы?»

РЕЗЮМЕ

- В реальном мире провалы криптографии, как правило, обусловлены тем, как именно она применяется. Нам уже известны хорошие примитивы и протоколы, которые можно использовать в большинстве случаев, поэтому основной причиной ошибок остается их неправильное применение.
- Для решения многих типичных проблем уже существуют специальные криптографические примитивы и протоколы. В большинстве случаев достаточно будет найти подходящую реализацию, позволяющую решить стоящую перед вами

задачу. Обязательно прочитайте руководство, чтобы понять, в каких ситуациях уместно использовать тот или иной примитив или протокол.

- Реальные протоколы строятся из криптографических примитивов, подобно конструктору Lego. Если вы столкнетесь с проблемой, для решения которой не существует проверенного протокола, вам придется комбинировать эти детали самостоятельно. Это очень опасно, так как криптографические примитивы иногда нарушаются при их применении в определенных ситуациях или в сочетании с другими примитивами или протоколами. В таких случаях формальная верификация оказывается отличным инструментом для поиска проблем, правда, использовать его бывает непросто.
- Реализация криптографии среди прочего осложняется необходимостью предоставления интерфейсов, устойчивых к злоупотреблениям (хороший криптографический код должен оставлять пользователю как можно меньше возможностей выстрелить себе в ногу).
- Чтобы избежать проблем в будущем, следует придерживаться консервативного подхода и задействовать проверенные криптографические инструменты. Проблемы, связанные со сложностью (например, с поддержкой слишком большого количества криптографических алгоритмов), широко обсуждаются в сообществе. Принцип отказа от использования чрезмерно сложных систем назван скучной криптографией. Будьте настолько скучными, насколько возможно.
- Криптографические примитивы и стандарты могут стать причиной появления ошибок в силу чрезмерной сложности их реализации или недостаточно четкого определения того, чего следует опасаться разработчикам. Концепция криптографического примитива или стандарта, плохо реализовать который оказывается довольно трудно, названа вежливой криптографией. Будьте вежливы.
- Применение криптографии в приложении иногда отдается на откуп пользователям. Удобная безопасность гарантирует то, что они понимают, как обращаться с криптографическим инструментом, и не могут им злоупотребить.
- Криптография — это не изолированный остров. Если вы будете следовать советам, которые я вам дал, то, скорее всего, большинство ошибок будет возникать в компонентах вашей системы, не связанных с криптографией. Не упускайте их из виду!
- Учитывая все то, что вы узнали из этой книги, будьте ответственны и хорошо продумывайте последствия своих действий.

Приложение
Ответы к упражнениям

ГЛАВА 2

Можете ли вы сказать, обеспечивает ли хеш-функция сокрытие и связывание, если она используется в качестве схемы обязательств?

Хеш-функция обеспечивает *сокрытие* благодаря свойству устойчивости к атаке нахождения прообраза при условии, что ваши входные данные достаточно случайны для того, чтобы никто не смог их угадать. Чтобы это исправить, вы можете сгенерировать случайное число и захешировать его с вашими входными данными, а затем *раскрыть* зафиксированные с помощью обязательства входные данные и случайное число. Хеш-функция обеспечивает *связывание* благодаря свойству устойчивости к атаке нахождения второго прообраза.

Кстати, такая строка никак не может представлять 256 бит (32 байта), верно? Как же тогда считать ее безопасной?

Нас не волнует устойчивость к коллизиям. Нас интересует только устойчивость к атаке нахождения второго прообраза. Таким образом, мы можем выполнить усечение дайджеста, чтобы уменьшить его размер.

Можете ли вы предположить, каким образом Dread Pirate Roberts (псевдоним администратора Silk Road) удалось получить хеш, содержащий название веб-сайта?

Dread Pirate Roberts создал множество ключей и в конце концов нашел такой, который хешируется в это классное представление в кодировке base32. Сервис Facebook сделал то же самое, благодаря чему стал доступен по адресу: facebookcorewwwi.onion (https://facebook.com/notes/protect-the-graph/making-connections-to-facebook-more-secure/1526085754298237). Такие адреса называются *тщеславными*.

ГЛАВА 3

Как вы думаете, каким образом счетчик переменной длины мог бы позволить злоумышленнику подделать аутентификационный тег?

Увидев следующее сообщение, в котором `||` обозначает конкатенацию строк, `MAC(k, "1" || "1 is my favorite number")`, злоумышленник может подделать действительный аутентификационный тег для одиннадцатого сообщения, `MAC(k, "11" | " is my favorite number")`.

Внимание: не все MAC являются PRF. Как вы думаете — почему?

Представьте, что функция `MAC(key, input)` представляет собой действительные MAC и PRF. Тогда является ли функция `NEW_MAC = MAC(key, input) || 0x01` действительным MAC? Является ли она действительной PRF? Данная функция представляет собой действительный MAC, так как предотвращает подделку, но она не является действительной PRF, поскольку вы можете легко отличить ее выход от абсолютно случайной строки (так как последний байт всегда установлен в значение 1).

ГЛАВА 6

Использовать один и тот же общий секрет для всех — плохое решение. Можете сказать почему?

Если с помощью этого общего секрета я могу зашифровать адресованные вам сообщения, то могу также расшифровывать сообщения, отправленные другими людьми.

Почему нельзя сразу использовать результат обмена ключами?

Вспомните, что вы узнали из главы 5, посвященной обмену ключами. В алгоритме (FF)DH вычисления выполняются по модулю большого простого числа p. Возьмем в качестве примера небольшое простое число 65 537. В шестнадцатеричной системе счисления это число p записывается в виде 0x010001, а в двоичной — 0000 0001 0000 0000 0000 0001. Обратите внимание на количество нулей, предшествующих первой единице в двоичном представлении. Оно объясняется тем, что мы представляем число в байтах, кратных 8 битам. Если вы разбираетесь в модульной арифметике, то знаете, что числа по модулю данного простого числа p никогда не будут больше, а значит, первые 7 бит всегда будут иметь значение 0. Кроме того, восьмой бит чаще всего будет установлен в значение 0, а не 1. В данном случае нельзя говорить о *равномерно распределенной случайной величине*. В идеале каждый бит должен принимать значение 1 или 0 с равной вероятностью.

ГЛАВА 7

Как вы видели в главе 3, аутентификационные теги, создаваемые MAC, должны проверяться за одно и то же время во избежание атак по времени. Как вы думаете, нужно ли делать то же самое для проверки подписей?

Нет, поскольку для проверки аутентификационного тега используется секретный ключ. Проверка подписи предполагает применение только открытого ключа, следовательно, ее не обязательно выполнять за постоянное время.

ГЛАВА 8

Представьте на минуту, что смешивание различных источников энтропии осуществляется простой функцией XOR. Как может измениться результат?

Источник энтропии с бэкдором может выполнить операцию XOR над всеми остальными источниками, сведя всю энтропию к 0.

Схемы подписи типа BLS, упомянутые на рис. 8.5 и в главе 7, создают уникальные подписи, но они не подходят для ECDSA и EdDSA. Почему?

В алгоритме ECDSA подписывающий может выбрать разные nonce, чтобы создать разные подписи для одной и той же пары ключей и сообщения. Хотя алгоритм цифровой подписи EdDSA выводит nonce детерминированным образом на основе подписываемого сообщения, это не означает, что подписывающий не может использовать любое nonce по своему выбору.

ГЛАВА 9

Компрометация закрытого ключа сервера в определенный момент будет иметь катастрофические последствия, поскольку злоумышленники, осуществляющие атаки посредника, смогут расшифровать все ранее записанные разговоры. Вы понимаете, как это может произойти?

Злоумышленник может отмотать историю назад и выдать себя за сервер в момент выполнения рукопожатия. Теперь у него есть закрытый ключ сервера. Вся остальная информация, необходимая для обмена ключами и формирования симметричных ключей после рукопожатия, является открытой.

Значения `signatureAlgorithm` *и* `signatureValue` *не содержатся в самом сертификате* `tbsCertificate`. *Знаете ли вы почему?*

Центр сертификации должен подписать сертификат, что приводит к парадоксу: подпись не может быть частью самой себя. Таким образом, ЦС должен

присоединить подпись к сертификату. В других стандартах и протоколах могут использоваться иные методы. Например, вы можете сделать подпись частью `tbsCertificate` и притвориться, что она состоит из одних только нулей, при подписании или проверке сертификата.

ГЛАВА 10

Знаете ли вы, почему содержимое электронного письма сжимается перед шифрованием, а не после?

Согласно определению шифра, шифротекст неотличим от случайной строки. В связи с этим алгоритмы сжатия не могут обнаружить закономерности в зашифрованных данных, позволяющие эффективно их сжать. По этой причине сжатие всегда выполняется перед шифрованием.

Можете ли вы придумать способ подписания сообщения, лишенный подобной двусмысленности?

Аутентифицируйте контекст. Для этого включите имена отправителя и получателя и их открытые ключи в подпись, а затем зашифруйте все это.

ГЛАВА 11

Иногда приложения пытаются решить проблему, связанную с тем, что сервер узнает пароль пользователя при регистрации, позволяя клиенту хешировать пароль перед отправкой на сервер. Можете ли вы определить, действительно ли это работает?

Отдельно взятое хеширование на стороне клиента не работает, как показала печально известная атака pass-the-hash (https://en.wikipedia.org/wiki/Pass_the_hash): если сервер хранит хешированный пароль Алисы, то любой, кто его украдет, сможет пройти аутентификацию, выдав себя за нее. Некоторые приложения выполняют хеширование как на стороне клиента, так и на стороне сервера, что в данном случае может помешать активному злоумышленнику узнать исходный пароль (правда, активный злоумышленник может отключить хеширование на стороне клиента, обновив код клиентского приложения).

Представьте себе протокол, в рамках которого требуется ввести правильный четырехзначный PIN-код для безопасного подключения к устройству. Какова вероятность выбрать правильный PIN-код простым угадыванием?

Вероятность правильно угадать PIN-код составляет 1:10 000. Вам бы здорово повезло, если бы у вас были такие шансы на выигрыш в лотерее.

Реальная криптография

Дэвид Вонг

Криптография — это фундамент информационной безопасности. Чтобы быть на шаг впереди хакеров, атакующих ваши системы, необходимо разбираться в инструментах, структурах и протоколах, защищающих ваши сети и приложения. В книге просто и наглядно описываются аутентификация, шифрование, подписи, хранение секретов и другие криптографические понятия.

Изучите приемы, которые используют разработчики, системные администраторы и специалисты по безопасности в своей повседневной работе. Вы не встретите ни сложных математических выкладок, ни умопомрачительного жаргона: современные криптографические методы рассматриваются с помощью понятных графиков и практических примеров. Познакомьтесь с хеш-функциями и подписями, криптографическими протоколами и безопасным обменом сообщениями, а также с передовыми достижениями, такими как постквантовая криптография и криптовалюты. Читать «Реальную криптографию» одно удовольствие, и, возможно, она спасет, когда в следующий раз злоумышленники покусятся на ваши данные.

- Реализация цифровых подписей и доказательств с нулевым разглашением информации.
- Специализированное оборудование для организации атак и враждебной среды.
- Выявление и устранение порочных практик.
- Выбор криптографического инструмента для решения конкретных задач.

Опыт в криптографии не требуется.

Дэвид Вонг — инженер по криптографии, активный участник разработки интернет-стандартов, включая протокол защиты транспортного уровня TLS.

«Содержит удивительно полный обзор современной криптографии. Незаменима для всех, кто начинает осваивать информационную безопасность».

— Томас Дойленд, Green Rocket Security

«Охватывает все самые важные аспекты прикладной криптографии. Эту книгу должен прочитать каждый разработчик».

— Харальд Кун, TurnFriendly

«Подробное введение в криптографию, охватывающее такие важные темы, как блокчейн и квантовые вычисления».

— Габор Ласло Хайба, ProLion

«Лучшая в своей категории книга. Она поможет вам пройти путь от любопытствующего новичка до уверенного практика».

— Уильям Руденмальм, Creandum

ПИТЕР®

WWW.PITER.COM
интернет-магазин

Заказ книг:
(812) 703-73-74
books@piter.com

PiterBooks
PiterForPeople
ThePiterBooks
Company/piter

MANNING

ISBN:978-5-4461-2091-8