О криптографии всерьез

Практическое введение в современное шифрование

Жан-Филипп Омассон

no starch press

ДМК ПРЕСС

Жан-Филипп Омассон

О криптографии всерьез

SERIOUS CRYPTOGRAPHY

A Practical Introduction to Modern Encryption

Jean-Philippe Aumasson

no starch press
San Francisco

О КРИПТОГРАФИИ ВСЕРЬЕЗ

Практическое введение в современное шифрование

Жан-Филипп Омассон

ДМК ПРЕСС
издательство
Москва, 2022

УДК 004.382
ББК 32.973-018
О57

Омассон Ж.-Ф.
О57 О криптографии всерьез / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 328 с.: ил.

ISBN 978-5-97060-975-0

В этом практическом руководстве по современному шифрованию анализируются фундаментальные математические идеи, лежащие в основе криптографии. Рассказывается о шифровании с аутентификацией, безопасной случайности, функциях хеширования, блочных шифрах и методах криптографии с открытым ключом, в частности RSA и криптографии на эллиптических кривых. Вы узнаете о том, как выбирать наилучший алгоритм или протокол, как избежать типичных ошибок безопасности и как задавать правильные вопросы поставщику. В заключительной части книги рассматриваются темы повышенной сложности, например TLS, а также обсуждается будущее криптографии в эпоху квантовых компьютеров.

В каждой главе приводятся примеры работы алгоритмов и материалы для дополнительного изучения.

Издание будет полезно как профессиональным разработчикам, так и тем, кто хочет разобраться в основах современной криптографии и успешно применять методы шифрования.

УДК 004.382
ББК 32.973-018

Title of English-language original: Serious Cryptography: A Practical Introduction to Modern Encryption, ISBN 9781593278267, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103.

ISBN 978-1-59327-826-7 (англ.)
ISBN 978-5-97060-975-0 (рус.)

СОДЕРЖАНИЕ

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте *www.dmkpress.com* на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

ВСТУПИТЕЛЬНОЕ СЛОВО

Если вы читали какие-нибудь книги по компьютерной безопасности, то, наверное, встречались с распространенным взглядом на криптографию. «Криптография, – говорят авторы, – самое прочное звено в цепочке». Оценка похвальная, но не вполне точная. Если криптография – действительно самая прочная часть системы, то зачем тратить время на ее улучшение, когда есть много других частей, нуждающихся во внимании?

И если попытаться сформулировать главный урок, который я хочу преподать этой книгой, то он состоит в том, что такой взгляд на криптографию, мягко говоря, идеализирован, а по сути является мифом. Криптография прочна *в теории*, а на практике столь же подвержена ошибкам, как и любой другой аспект системы безопасности. Особенно это относится к случаям, когда криптографическую систему реализуют непрофессионалы, не имеющие достаточного опыта и пренебрегающие деталями, – а таковы многие развернутые в настоящее время системы. Хуже того – если реализация криптографической системы содержит ошибки, то проявления их зачастую оказываются весьма скандальными.

Но какое вам до этого дело, и зачем нужна эта книга?

Когда почти двадцать лет назад я только начинал заниматься прикладной криптографией, разработчикам ПО была доступна лишь отрывочная и не самая актуальная информация. Криптографы придумывали алгоритмы и протоколы, а программисты, реализовывая их, создавали запутанные, плохо документированные библиотеки, пред-

назначенные в первую очередь для других специалистов. Существовало – и до сих пор существует – глубокая пропасть между теми, кто знает и понимает криптографические алгоритмы, и теми, кто этими алгоритмами пользуется (или на свой страх и риск пренебрегает ими). На рынке имеется очень немного достойных книг, и еще меньше полезных программисту-практику.

Результаты такого положения удручают. Я имею в виду факты компрометации, о которых говорят такие ярлыки, как «CVE» или «Серьезность: высокая», а в некоторых особо тревожных ситуациях – атаки, которые на слайдах снабжены грифом «СОВЕРШЕННО СЕКРЕТНО». Если вам и известны некоторые особо знаменитые примеры, то только потому, что они затрагивают системы, от которых зависит ваша работа. Многие проблемы подобного рода возникают, потому что криптография – сложная и математически элегантная теория, а специалисты по ней не потрудились поделиться знаниями с инженерами, которые пишут программный код.

По счастью, ситуация начинает меняться, и пример тому – эта книга.

Книга «О криптографии всерьез» написана одним из самых известных экспертов по прикладной криптографии, однако ориентирована не на других специалистов того же профиля. Но и поверхностным обзором этой дисциплины она тоже не является. Напротив, она содержит скрупулезное и современное обсуждение криптографической техники, призванное помочь стать лучше практикам, собирающимся подвизаться в этой области. Вы узнаете не только о том, как работают алгоритмы, но и как использовать их в реальных системах.

Изложение начинается с рассмотрения основных криптографических примитивов, в т. ч. блочных шифров, схем шифрования с открытым ключом, функций хеширования и генераторов случайных чисел. В каждой главе приводятся примеры работы алгоритмов и объясняется, что следует и чего *не* следует делать. В последних главах рассматриваются темы повышенной трудности, например TLS, а также будущее криптографии – что делать после того, как придут квантовые компьютеры и усложнят нам жизнь.

Конечно, одна книга не в состоянии решить все наши проблемы, но знания накапливаются по крупицам. В этой книге таких крупиц много. Быть может, даже достаточно для того, чтобы развернутые на практике криптографические системы стали, наконец, соответствовать тем высоким требованиям, которые к ним предъявляются.

Полезного вам чтения.

Мэттью Д. Грин,
профессор Института информационной безопасности
университета Джонса Хопкинса

ПРЕДИСЛОВИЕ

Я написал книгу, которую хотел бы иметь, когда только начал изучать криптографию. В 2005 году под Парижем я учился в магистратуре и с предвкушением записался на спецкурс по криптографии. Увы, спецкурс был отменен, потому что не набралось достаточного количества желающих. «Криптография – это слишком сложно», – говорили студенты и массово записывались на курсы по компьютерной графике и базам данных.

С тех пор слова «криптография – это сложно» я слышал неоднократно. Но так ли сложна криптография в действительности? Чтобы играть на музыкальном инструменте, овладеть языком программирования или воплотить на практике достижения какой-нибудь увлекательной дисциплины, необходимо изучить определенные концепции и символы, но для этого необязательно иметь докторскую степень. Я думаю, что это относится и к желанию стать компетентным криптографом. А также полагаю, что криптография считается такой трудной наукой, потому что криптографы не приложили достаточных усилий к ее преподаванию.

И есть еще одна причина, по которой я считаю эту книгу необходимой: криптография превратилась в довольно разветвленную область знаний. Чтобы сделать что-то полезное и важное в криптографии, нужно разбираться в *смежных* областях: как работают компьютеры и сети, что нужно пользователям и системам и как противник может злонамеренно воспользоваться алгоритмами и их реализациями. Иными словами, необходима связь с реальностью.

Подход, принятый в этой книге

Первоначально книга называлась «Crypto for Real», чтобы подчеркнуть практически ориентированный, деловой подход, которому я намеревался следовать. Я хотел не столько опростить криптографию, сколько связать с ее с реальными приложениями. Я привожу примеры исходного кода и описываю реальные ошибки и кошмарные истории.

Помимо четкой связи с реальностью, в основу книги положены еще два краеугольных камня: простота и современность. Упрощаю я изложение только по форме, не жертвуя содержанием: представляю много нетривиальных идей, но без скучного математического формализма. Я хочу, чтобы читатель понял основополагающие идеи криптографии, это кажется мне важнее, чем запоминание бесконечных формул. Что касается современности, то я рассматриваю последние теоретические результаты и приложения, например протокол TLS 1.3 и криптографию в постквантовую эпоху. Я не обсуждаю детали устаревших или небезопасных алгоритмов, например DES или MD5. Исключением является алгоритм RC4, но и он включен только для того, чтобы продемонстрировать, в чем его слабость и как работают потоковые шифры такого рода.

Книга «Криптография всерьез» не является ни путеводителем по криптографическому программному обеспечению, ни сводом технических спецификаций – такие вещи легко найти в сети. Ее главное назначение – пробудить у вас интерес к криптографии и попутно рассказать о ее фундаментальных концепциях.

Для кого предназначена эта книга

Во время работы над книгой я часто представлял себе читателя как разработчика, который оказался вынужден иметь дело с криптографией, но так и остался в растерянности после чтения заумных учебников и научных статей. Разработчики часто должны – и хотят – лучше понимать криптографию, чтобы избежать неудачных проектных решений, и я надеюсь, что в этом моя книга поможет.

Но если вы не являетесь разработчиком, тоже ничего страшного! Чтение книги не потребует от вас владения навыками кодирования, она доступна любому, кто знаком с основами информатики и математикой в объеме технического вуза (начала теории вероятностей, арифметики по модулю и т. д.).

Но, несмотря на сравнительную доступность книги, для получения от нее максимальной пользы все же требуется приложить некоторые усилия. Мне приходит на ум аналогия с альпинизмом: автор прокладывает путь, снабжает вас веревками и ледорубом, но покорить гору вам придется самостоятельно. Изучение изложенных в книге идей потребует труда, но в конце преодолевшего все препятствия ждет награда.

Структура книги

Книга состоит из четырнадцати глав, разбитых на четыре части. По большей части главы независимы, за исключением главы 9, в которой заложен фундамент для трех последующих глав. Но я все же рекомендую сначала прочитать первые три главы.

Основы

- **Глава 1 «Шифрование».** Здесь вводится понятие безопасного шифрования, начиная со слабых шифров с использованием карандаша и бумаги и заканчивая стойкими рандомизированными шифрами.
- **Глава 2 «Случайность».** Описывается, как работает генератор псевдослучайных чисел, когда такой генератор считается безопасным и как его безопасно использовать.
- **Глава 3 «Криптографическая безопасность».** Обсуждается теоретическая и практическая безопасность, сравнивается предположительная и доказуемая безопасность.

Симметричные криптографические системы

- **Глава 4 «Блочные шифры».** Рассматриваются шифры, обрабатывающие сообщения поблочно. Основное внимание уделяется самому известному из них, Advanced Encryption Standard (AES).
- **Глава 5 «Потоковые шифры».** Описываются шифры, порождающие поток случайных на первый взгляд битов, которые объединяются с сообщением операцией XOR.
- **Глава 6 «Функции хеширования».** Функции хеширования – чуть ли не единственный алгоритм, не нуждающийся в секретном ключе, и при этом один из самых распространенных строительных блоков в криптографии.
- **Глава 7 «Хеширование с секретным ключом».** Объясняется, что будет, если соединить функцию хеширования с секретным ключом, и как этим можно воспользоваться для аутентификации сообщений.
- **Глава 8 «Шифрование с аутентификацией».** На примерах описываются алгоритмы, которые могут одновременно зашифровать и аутентифицировать сообщение, в частности стандарт AES-GCM.

Асимметричные криптографические системы

- **Глава 9 «Трудные задачи».** Здесь заложен теоретический фундамент шифрования с открытым ключом; используется нотация из теории вычислительной сложности.
- **Глава 10 «RSA».** В алгоритме RSA задача разложения на множители применяется для построения схем безопасного шифрования и цифровой подписи с помощью простых арифметических операций.

- **Глава 11 «Протокол Диффи–Хеллмана».** Идея асимметричной криптографии обобщается на понятие совместной выработки ключей, когда две стороны вырабатывают секретное значение, используя только несекретные данные.
- **Глава 12 «Эллиптические кривые».** Простое введение в эллиптическую криптографию – самый быстрый вид асимметричных криптографических систем.

Приложения

- **Глава 13 «Протокол TLS».** Рассматривается протокол Transport Layer Security (TLS), пожалуй, самый важный для безопасности сетей.
- **Глава 14 «Квантовая и постквантовая криптография».** В этой заключительной главе с оттенком научной фантастики обсуждаются квантовые вычисления и новый вид криптографии.

Благодарности

Хочу поблагодарить Яна, Энни и других сотрудников издательства No Starch, принявших участие в подготовке этой книги, а особенно Билла, который поверил в проект с самого начала, терпеливо усваивал трудные темы и превращал мои беспорядочные черновики в читаемый текст. Я также благодарен Лорел, которая вносила мои многочисленные поправки и благодаря которой книга выглядит так симпатично.

Что касается технической стороны, то книга содержала бы куда больше ошибок и неточностей, если бы не помощь следующих лиц: Джон Каллас, Билл Кокс, Нильс Фергюсон, Филипп Йованович, Сэмюэл Нивс, Дэвид Рейд, Филлип Рогуэей, Эрик Тьюз, – а также читателей предварительной версии, сообщавших о найденных ошибках. Наконец, я благодарю Мэтта Грина, написавшего вступительное слово.

Я также выражаю благодарность своему работодателю, компании Kudelski Security, выделившей мне время для работы над книгой. Наконец, моя глубочайшая благодарность Александре и Мелине за поддержку и терпение.

Лозанна, 17.05.2017 (три простых числа)

СПИСОК АББРЕВИАТУР

AE	authenticated encryption (шифрование с аутентификацией)
AEAD	authentication encryption with associated data (шифрование с аутентификацией и ассоциированными данными)
AES	Advanced Encryption Standard (улучшенный стандарт шифрования)
AES-NI	AES native instructions (AES с платформенными командами)
AKA	authenticated key agreement (совместная выработка ключей с аутентификацией)
API	application program interface (интерфейс прикладной программы)
ARX	add-rotate-XOR
ASIC	application-specific integrated circuit (специализированная заказная интегральная схема)
CA	certificate authority (удостоверяющий центр, УЦ)
CAESAR	Конкурс шифрования с аутентификацией: безопасность, применимость и надежность
CBC	cipher block chaining (режим сцепления блоков шифртекста)
CCA	chosen-ciphertext attack (атака с подобранным шифртекстом)
CDH	computational Diffie–Hellman (предположение о вычислительной трудности задачи Диффи–Хеллмана)
CMAC	cipher-based MAC (имитовставка на основе блочного шифра)
COA	ciphertext-only attack (атака на основе шифртекста)
CPA	chosen-plaintext attack (атака с подобранным открытым текстом)
CRT	Chinese remainder theorem (китайская теорема об остатках)
CTR	режим счетчика
CVP	closest vector problem (задача о ближайшем векторе)
DDH	decisional Diffie–Hellman (предположение Диффи–Хеллмана о распознавании)

DES	Data Encryption Standard (стандарт шифрования данных)
DH	Diffie–Hellman
DLP	discrete logarithm problem (задача дискретного логарифмирования)
DRBG	deterministic random bit generator (детерминированный генератор случайных битов)
ECB	electronic codebook (режим электронной кодовой книги)
ECC	elliptic curve cryptography (эллиптическая криптография)
ECDH	elliptic curve Diffie–Hellman (эллиптический метод Диффи–Хеллмана)
ECDLP	elliptic-curve discrete logarithm problem (задача дискретного логарифмирования на эллиптической кривой)
ECDSA	elliptic-curve digital signature algorithm (алгоритм цифровой подписи на эллиптической кривой)
FDH	Full Domain Hash (полный хеш домена)
FHE	fully homomorphic encryption (полностью гомоморфное шифрование)
FIPS	Federal Information Processing Standards (Федеральный стандарт обработки информации)
FPE	format-preserving encryption (шифрование с сохранением формата)
FPGA	field-programmable gate array (программируемая пользователем вентильная матрица, ППВМ)
FSR	feedback shift register (регистр сдвига с обратной связью)
GCD	greatest common divisor (наибольший общий делитель, НОД)
GCM	Galois Counter Mode (режим счетчика с аутентификацией Галуа)
GNFS	general number field sieve (общий метод решета числового поля)
HKDF	HMAC-based key derivation function (функция формирования ключа на основе HMAC)
HMAC	hash-based message authentication code (имитовставка на основе функции хеширования)
HTTPS	HTTP Secure (безопасный HTTP)
IND	indistinguishablity (неразличимость)
IP	Internet Protocol
IV	initial value (начальное значение)
KDF	key derivation function (функция формирования ключа)
KPA	known-plaintext attack (атака с известным простым текстом)
LFSR	linear feedback shift register (линейный регистр сдвига с обратной связью)
LSB	least significant bit (младший бит)
LWE	learning with errors (обучение с ошибками)
MAC	messsage authentication code (имитовставка)
MD	message digest (дайджест сообщения)
MitM	meet-in-the-middle (метод встречи посередине)
MQ	multivariate quadratics (многомерные системы квадратичных уравнений)
MQV	Menezes–Qu–Vanstone (протокол Менезеса–Кью–Вэнстоуна)
MSB	most significant bit (старший бит)

MT	Mersenne Twister (вихрь Мерсенна)
NFSR	nonlinear feedback shift register (нелинейный регистр сдвига с обратной связью)
NIST	National Institute of Standards and Technology (Национальный институт стандартов и технологий)
NM	non-malleability (неподатливость)
OAEP	Optimal Asymmetric Encryption Padding (оптимальное асимметричное шифрование с дополнением)
OCB	offset codebook (режим кодовой книги со смещением)
P	polynomial time (полиномиальное время)
PLD	programmable logic device (программируемое логическое устройство)
PRF	pseudorandom function (псевдослучайная функция)
PRNG	pseudorandom number generator (генератор псевдослучайных чисел)
PRP	pseudorandom permutation (псевдослучайная перестановка)
PSK	pre-shared key (предварительно разделенный ключ)
PSS	Probabilistic Signature Scheme (вероятностная схема подписи)
QR	quarter-round (четверть раунда)
QRNG	quantum random number (квантовый генератор случайных чисел)
RFC	request for comments (запрос на комментарии)
RNG	random number generator (генератор случайных чисел)
RSA	Rivest–Shamir–Adleman (алгоритм Ривеста–Шамира–Адлемана)
SHA	Secure Hash Algorithm (безопасный алгоритм хеширования)
SIS	short integer solution (короткое целочисленное решение)
SIV	synthetic IV (синтетическое начальное значение)
SPN	substitution–permutation network (подстановочно-перестановочная сеть)
SSH	Secure Shell (безопасная оболочка)
SSL	Secure Socket Layer (уровень безопасных сокетов)
TE	tweakable encryption (настраиваемое шифрование)
TLS	Transport Layer Security (безопасность транспортного уровня)
TMTO	time-memory trade-off (компромисс между временем и памятью)
UDP	User Datagram Protocol (протокол пользовательских дейтаграмм)
UH	universal hash (универсальная функция хеширования)
WEP	Wired Equivalent Privacy (протокол безопасности в беспроводных сетях)
WOTS	Winternitz one-time signature (одноразовая подпись Винтерница)
XOR	exclusive OR (исключающее ИЛИ)

1

ШИФРОВАНИЕ

Шифрование – главное применение криптографии; его цель – сделать данные непонятными и тем самым обеспечить их *конфиденциальность*. Для шифрования используется алгоритм, называемый *шифром*, и секретное значение, называемое *ключом*; не зная секретного ключа, невозможно получить ни одного бита зашифрованного сообщения, не говоря уже о том, чтобы его дешифрировать.

В данной главе предметом нашего внимания будет *симметричное шифрование*, его простейшая разновидность. В этом случае для дешифрирования используется тот же ключ, что для шифрования (в отличие от *асимметричного шифрования*, или *шифрования с открытым ключом*, когда ключи шифрования и дешифрирования различны). Мы начнем с рассмотрения самых слабых форм симметричного шифрования – классических шифров, способных устоять только перед совсем необразованным противником, а затем перейдем к самым стойким шифрам, взломать которые вообще невозможно.

Основы

В контексте шифрования *открытым текстом* называется незашифрованное сообщение, а *шифртекстом* – зашифрованное сообщение. Шифр состоит из двух функций: *шифрование* преобразует открытый текст в шифртекст, а *дешифрирование* производит обратное преобразование шифртекста в открытый. Но часто говорят «шифр», имея в виду «шифрование». Например, на рис. 1.1 показан шифр **E**, представленный прямоугольником, который принимает открытый текст *P* и ключ *K* и порождает на выходе шифртекст *C*. Я буду записывать это соотношение в виде $C = \mathbf{E}(K, P)$. Аналогично, когда шифр работает в режиме дешифрирования, я буду писать $\mathbf{D}(K, C)$.

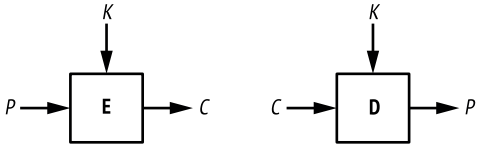

Рис. 1.1. Простейшее шифрование и дешифрирование

Примечание *Для некоторых шифров размер шифртекста равен размеру открытого текста, для других он немного больше. Но никогда шифртекст не может быть короче открытого текста.*

Классические шифры

Классическими называются шифры, появившиеся раньше компьютеров и потому применяемые к буквам, а не к битам. Они гораздо проще современных шифров типа DES – например, в Древнем Риме или во время Первой мировой войны для шифрования сообщения нельзя было воспользоваться всей мощью компьютера, так что приходилось довольствоваться бумагой и карандашом. Существует много классических шифров, но наиболее известны шифры Цезаря и Виженера.

Шифр Цезаря

Шифр Цезаря назван так, потому что, согласно древнеримскому историку Светонию, им пользовался Юлий Цезарь. Сообщение шифруется путем сдвига каждой буквы на три позиции вправо по алфавиту с оборотом по достижении Z. Например, ZOO шифруется как CRR, результатом дешифрирования FDHVDU является CAESAR и т. д., как показано на рис. 1.2. В числе 3 нет ничего особенного, просто так производить вычисления в уме проще, чем при выборе, скажем, 11 или 23.

Взломать шифр Цезаря проще простого: чтобы дешифрировать заданный шифртекст, нужно просто сдвинуть каждую букву на три позиции влево. Однако шифр Цезаря, наверное, был достаточно стойким во времена Красса и Цицерона. Поскольку никакого секретного ключа нет

(он всегда равен 3), пользователи шифра Цезаря должны были считать, что противник тупой и не знает, как его найти, – такое предположение в наши дни выглядит совсем уж нереалистично. (Правда, в 2006 году итальянская полиция арестовала босса мафии, расшифровав сообщения, написанные на клочках бумаги с использованием варианта шифра Цезаря; например, ABC было зашифровано как 456, а не DEF.)

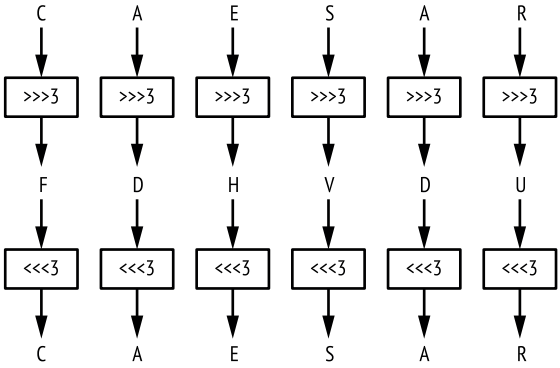

Рис. 1.2. Шифр Цезаря

Можно ли сделать шифр Цезаря более безопасным? Например, можно было сдвигать не на 3, а на какое-нибудь секретное значение, но это не спасало бы положения, потому что противнику достаточно было бы перебрать все 25 возможных значений сдвига и найти то, при котором расшифрованное сообщение становится осмысленным.

Шифр Виженера

Для существенного улучшения шифра Цезаря потребовалось примерно 1500 лет, в XVI веке итальянец Джовани Баттиста Белассо создал шифр Виженера. Имя «Виженер» принадлежит французу Блезу де Виженеру, который изобрел в XVI веке другой шифр, но из-за неправильной атрибуции вошел в историю. Так или иначе, шифр Виженера обрел популярность и использовался, в частности, конфедератами во время Гражданской войны в США и швейцарской армией во время Первой мировой войны.

Шифр Виженера похож на шифр Цезаря, только величина сдвига составляет не три позиции, а определяется *ключом*, набором букв, которым соответствуют числа, равные позиции буквы в алфавите. Например, если ключ равен DUH, то буквы открытого текста сдвигаются на 3, 20 и 7 позиций, потому что *D* отстоит от *A* на три позиции, *U* – на 20 позиций, а *H* – на семь позиций. Последовательность 3, 20, 7 повторяется, пока не будет зашифрован весь открытый текст. Например, слово CRYPTO на ключе DUH было бы зашифровано как FLFSNV: *C* сдвигается на три позиции и превращается в *F*, *R* сдвигается на 20 и превращается в *L* и т. д. На рис. 1.3 показано, как этот принцип применяется к шифрованию предложения THEY DRINK THE TEA.

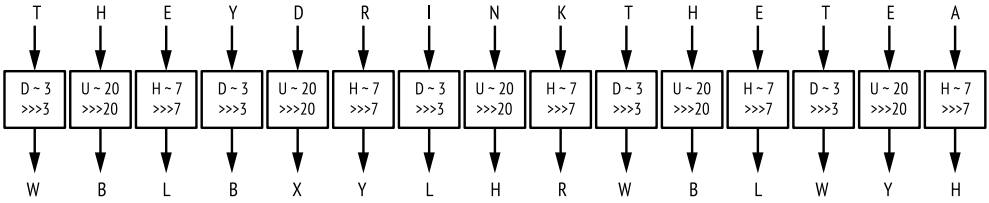

Рис. 1.3. Шифр Виженера

Очевидно, что шифр Виженера безопаснее, чем шифр Цезаря, но все равно его довольно легко взломать. Первый шаг взлома – определить длину ключа. Например, рассмотрим рис. 1.3, где результатом шифрования фразы THEY DRINK THE TEA с ключом DUH является строка WBLBXYLHRWBLWYH (пробелы обычно удаляются, чтобы не раскрывать границы слов). Заметим, что в шифртексте WBLBXYLHRWBLWYH группа из трех букв WBL встречается дважды с интервалом в девять букв. Это позволяет предположить, что было зашифровано одно и то же трехбуквенное слово с одинаковыми величинами сдвига. Поэтому криптоаналитик может сделать вывод, что длина ключа либо равна девяти, либо является делителем девяти (т. е. равна трем). Кроме того, он может догадаться, что повторяющееся трехбуквенное слово – THE, а значит, DUH – возможный ключ шифрования.

Второй шаг взлома шифра Виженера – определение самого ключа методом *частотного анализа*, в котором используется тот факт, что распределение букв в естественных языках неравномерно. Например, в английском чаще всего встречается буква *E*, поэтому, обнаружив, что в шифртексте чаще других встречается *X*, мы можем заключить, что в соответствующей позиции открытого текста, скорее всего, находится буква *E*.

Несмотря на относительную слабость, шифр Виженера, наверное, был достаточно хорош в те времена, когда применялся. Во-первых, для описанной выше атаки нужны сообщения, содержащие хотя бы несколько предложений, поэтому она не будет работать, если шифруются только короткие сообщения. Во-вторых, большая часть сообщений должна была сохранять секретность только в течение короткого промежутка времени, так что даже если в конечном итоге враг их расшифрует, это уже не будет иметь значения. (В XIX веке криптограф Огюст Керкгоффс оценил, что для большинства зашифрованных военных сообщений конфиденциальность требовалась лишь на протяжении трех-четырех часов.)

Как работают шифры

Проанализировав простые шифры, в частности Цезаря и Виженера, мы можем попытаться выделить общие принципы работы шифра и прежде всего два основных компонента: перестановка и режим работы. *Перестановка* – это функция, которая преобразует элемент (в криптографии букву или группу битов) таким образом, что для

каждого элемента однозначно определено обратное преобразование (например, в случае шифра Цезаря это сдвиг на три буквы). *Режим работы* – это алгоритм, который использует перестановку для обработки сообщений произвольного размера. Режим работы шифра Цезаря тривиален – одна и та же перестановка повторяется для каждой буквы. Но в шифре Виженера режим работы сложнее – буквы в разных позициях подвергаются разным перестановкам.

В следующих разделах я более подробно разберу эти компоненты и их связь с безопасностью шифра. Я покажу, почему классические шифры принципиально небезопасны, в отличие от современных шифров, исполняемых высокопроизводительными компьютерами.

Перестановка

В большинстве классических шифров одна буква просто заменяется другой, т. е. мы выполняем *подстановку*. В шифрах Цезаря и Виженера подстановка – это сдвиг в алфавите, хотя сам алфавит или набор символов может меняться: вместо английского языка может быть арабский, а вместо букв – слова, числа или идеограммы. Представление, или кодирование, информации – отдельная тема, слабо связанная с безопасностью. (Мы рассматриваем латинские буквы просто потому, что так использовались классические шифры.)

Подстановка в шифре не может быть произвольной. Это должна быть перестановка букв от *A* до *Z*, при которой для каждой буквы однозначно определена обратная ей. Например, подстановка, преобразующая буквы *A*, *B*, *C*, *D* соответственно в *C*, *A*, *D*, *B*, является перестановкой, поскольку на каждую букву отображается ровно одна буква. Но подстановка, преобразующая *A*, *B*, *C*, *D* в *D*, *A*, *A*, *C*, перестановкой не является, потому что обе буквы *B* и *C* отображаются в *A*, а в случае перестановки у каждой буквы должен быть ровно один прообраз. Кроме того, не каждая перестановка безопасна. Чтобы считаться безопасной, перестановка шифра должна удовлетворять трем условиям.

- **Перестановка должна определяться ключом**, это позволит держать ее в секрете, пока ключ не скомпрометирован. В шифре Виженера, не зная ключа, невозможно сказать, какая из 26 перестановок использовалась, поэтому взломать его не слишком просто.
- **Разным ключам должны соответствовать разные перестановки.** В противном случае будет проще расшифровать сообщение, не зная ключа: если разные ключи порождают одинаковые перестановки, то различных ключей меньше, чем перестановок, и, значит, для дешифрирования без ключа нужно будет перебрать меньше вариантов. В шифре Виженера каждая буква ключа определяет подстановку; всего существует 26 разных букв и столько же различных перестановок.
- **Перестановка должна выглядеть случайной (формальные детали опустим).** После применения перестановки в шифртекс-

те не должно быть никаких закономерностей, поскольку наличие таковых позволит противнику предсказать перестановку, а значит, уровень безопасности снизится. Например, в шифре Виженера подстановка в достаточной мере предсказуема: определив, что *A* преобразуется в *F*, можно было бы заключить, что величина сдвига равна 5, и тогда мы знали бы, что *B* преобразуется в *G*, *C* в *H* и т. д. Но если перестановка выбирается случайно, то, зная, что шифр преобразует *A* в *F*, мы могли бы только сказать, что он *не* преобразует *B* в *F*.

Перестановки, удовлетворяющие этим условиям, мы будем называть *безопасными*. Но далее мы увидим, что безопасность перестановки – необходимое, но еще недостаточное условие для построения безопасного шифра. Необходим еще режим работы шифра, чтобы поддержать сообщения произвольной длины.

Режим работы

Пусть имеется безопасная перестановка, преобразующая, например, *A* в *X*, *B* в *M* и *N* в *L*. Слово BANANA тогда будет преобразовано в MXLXLX, в котором каждая буква *A* заменена на *X*. Таким образом, использование одной и той же перестановки для всех букв открытого текста показывает, какие буквы повторяются. Путем анализа повторений вы, возможно, не раскроете все сообщение целиком, но *что-то* о нем узнаете. В примере слова BANANA не нужно знать ключ, чтобы догадаться, что в тех позициях, где в зашифрованном тексте мы видим *X*, в открытом тексте находятся одинаковые буквы. И то же самое относится к двум позициям, занятым буквой *L*. Так что если заранее известно, к примеру, что сообщение содержит название фрукта, то можно быть уверенным, что это не CHERRY, LYCHEE или другой фрукт из шести букв, а, скорее всего, BANANA.

Режим работы (или просто *режим*) шифра уменьшает шансы на раскрытие повторяющихся букв в открытом тексте, потому что для них используются разные перестановки. Режим шифра Виженера решает эту проблему частично: если ключ состоит из *N* букв, то для каждой из *N* последовательных букв будет использована своя перестановка. Но это все же не исключает появления закономерностей в шифртексте, потому что для каждой *N*-й буквы сообщения используется одна и та же перестановка. Именно поэтому шифр Виженера можно взломать с помощью частотного анализа.

В шифре Виженера от частотного анализа можно защититься, если шифровать только открытые тексты, длина которых совпадает с длиной ключа. Но все равно остается другая проблема: при многократном использовании одного и того же ключа не скрыть сходство открытых текстов. Например, если ключ равен KYN, то слова TIE и PIE преобразуются в DGR и ZGR соответственно. Оба шифртекста заканчиваются двумя одинаковыми буквами (*GR*), это говорит о том, что и в обоих открытых текстах две последние буквы совпадают. Безопасный шифр не

должен раскрывать таких закономерностей. Для построения безопасного шифра нужно сочетать безопасную перестановку с безопасным режимом. В идеале их комбинация не должна позволить противнику узнать о сообщении что-то, кроме его длины.

Почему классические шифры небезопасны

Классические шифры обречены оставаться небезопасными, потому что мы ограничены операциями, которые можно выполнить в уме или на бумаге. Им недостает вычислительной мощности компьютера, поэтому они легко взламываются с помощью простых программ. Обсудим фундаментальную причину, по которой в современном мире простота делает такие шифры небезопасными.

Напомним, что перестановка шифра должна выглядеть случайной, чтобы считаться безопасной. Конечно, лучший способ выглядеть случайной – это *быть* случайной, т. е. мы должны случайным образом выбирать каждую перестановку из множества всех возможных. А выбирать есть из чего. Для английского алфавита из 26 букв имеется приблизительно 2^{88} перестановок:

$$26! = 403291461126605635584000000 \approx 2^{88}.$$

Здесь восклицательный знак (!) – символ факториала, определяемого следующим образом:

$$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2.$$

(Чтобы понять, откуда берется это число, будем подсчитывать перестановки как упорядоченные списки букв: существует 26 способов выбрать первую букву, после чего вторую букву можно выбрать 25 способами, третью – 24 способами и т. д.) Это число поистине огромно; по порядку величины оно такое же, как число атомов в теле человека. Но для классических шифров доступна лишь малая доля этого множества перестановок, а именно те, для вычисления которых достаточно простых операций (например, сдвигов) и которые имеют простое описание (например, простой алгоритм или небольшую таблицу преобразования). Проблема в том, что перестановка, удовлетворяющая обоим ограничениям, не может быть безопасной.

Безопасные перестановки можно получить с помощью простых операций, выбрав случайную перестановку из множества, представленного в виде таблицы, содержащей 25 букв (достаточно, чтобы представить перестановку 26 букв, из которых двадцать шестая отсутствует), а для ее применения нужно будет найти буквы в этой таблице. Но тогда описание не будет коротким. Например, чтобы описать 10 разных перестановок, потребуется 250 букв, а не 10, как в шифре Виженера.

Можно также получить безопасные перестановки с коротким описанием, если не ограничиваться сдвигами по алфавиту, а добавить

более сложные операции, например сложение, умножение и т. д. Так работают современные шифры: имея ключ длиной 128 или 256 бит, они выполняют сотни поразрядных операций для шифрования одной-единственной буквы. На компьютере, выполняющем миллиарды поразрядных операций в секунду, это не занимает много времени, но на вычисления вручную пришлось бы потратить несколько часов, и все равно результат был бы уязвим для частотного анализа.

Идеальное шифрование: одноразовый блокнот

В принципе, классический шифр не может быть безопасным, если только не используется очень длинный ключ, но шифровать гигантским ключом практически неудобно. Однако такой шифр существует, он называется одноразовым блокнотом и является самым безопасным. По существу, он гарантирует *идеальную секретность*: даже располагая неограниченной вычислительной мощностью, противник не сможет узнать об открытом тексте ничего, кроме длины.

В следующих разделах я покажу, как работает одноразовый блокнот, а затем приведу набросок доказательства его безопасности.

Шифрование с помощью одноразового блокнота

Одноразовый блокнот принимает открытый текст P и случайный ключ K такой же длины, как P, и порождает шифртекст C, вычисляемый по формуле

$$C = P \oplus K,$$

где C, P, K – битовые строки одинаковой длины, а $\oplus$ – операция поразрядного исключающего ИЛИ (XOR), определенная следующим образом: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$.

Примечание *Я описываю одноразовый блокнот в стандартной форме, применяемой к битам, но его можно адаптировать и для других символов. Например, в случае букв мы получим вариант шифра Цезаря, в котором величина сдвига каждой буквы выбирается случайным образом.*

Дешифрирование с одноразовым блокнотом выполняется точно так же, как шифрование; это просто операция XOR: $P = C \oplus K$. Действительно, легко проверить, что $C \oplus K = P \oplus K \oplus K = P$, поскольку $K \oplus K$ дает строку, состоящую из одних нулей 000 … 000. Вот так – даже проще, чем шифр Цезаря.

Например, если P = 01101101 и K = 10110100, то вычисление выглядит следующим образом:

$$C = P \oplus K = 01101101 \oplus 10110100 = 11011001.$$

Дешифрирование дает P в результате такого вычисления:

$$P = C \oplus K = 11011001 \oplus 10110100 = 01101101.$$

Важно отметить, что одноразовый блокнот можно использовать только *один раз*: каждый ключ K следует использовать лишь однократно. Если бы один и тот же K использовался для зашифровывания P_1 и P_2 в C_1 и C_2, то пассивный противник мог бы подслушать и вычислить выражение:

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2 \oplus K \oplus K = P_1 \oplus P_2.$$

Тогда пассивный противник узнал бы результат применения XOR к P_1 и P_2, хотя эта информация должна держаться в секрете. Более того, если хотя бы одно открытое сообщение известно, то можно восстановить и другое.

Разумеется, одноразовый блокнот крайне неудобно использовать, потому что требуется ключ такой же длины, как простой текст, да еще для каждого нового сообщения или группы данных необходимо выбирать новый случайный ключ. Чтобы зашифровать терабайтный жесткий диск, понадобился бы другой диск такого же объема для хранения ключа! И тем не менее одноразовые блокноты использовались на практике. Например, их применяло Британское управление специальных операций во время Второй мировой войны, сотрудники КГБ и АНБ, да и сейчас они используются в особых ситуациях. (Я слышал историю о швейцарских банкирах, которые не смогли договориться о шифре, которому доверяли бы обе стороны, и потому остановились на одноразовых блокнотах, но поступать так не рекомендую.)

Почему одноразовый блокнот безопасен?

Хотя одноразовый блокнот непрактичен, важно понимать, почему именно он безопасен. В 1940-х годах американский математик Клод Шеннон доказал, что для достижения идеальной секретности ключ одноразового блокнота должен быть не короче сообщения. Идея доказательства довольно проста. Предположим, что противник обладает неограниченными возможностями и, стало быть, может перепробовать все ключи. Наша цель – зашифровать сообщение так, чтобы противник, располагающий шифртекстом, не мог исключить ни одного возможного открытого текста.

Интуитивно свойство одноразового блокнота, обеспечивающее идеальную секретность, можно описать так: если ключ K случайный, то результирующее сообщение C представляется противнику таким же случайным, как K, потому что применение XOR к случайной строке и произвольной фиксированной строке дает случайную строку. Чтобы убедиться в этом, рассмотрим вероятность получения 0 в первом бите случайной строки (она равна 1/2). Какова вероятность, что результат применения XOR к случайному биту и второму биту будет равен 0?

Правильно, снова 1/2. То же рассуждение можно распространить на битовые строки произвольной длины. Таким образом, шифртекст *C* кажется случайным противнику, не знающему *K*, поэтому действительно невозможно узнать что-то о *P* по *C*, даже при наличии у противника неограниченного времени и вычислительной мощности. Иными словами, знание шифртекста не дает никакой информации об открытом тексте, кроме его длины, а это и есть определение безопасного шифра.

Например, если длина шифртекста равна 128 бит (а это значит, что и длина открытого текста составляет 128 бит), то количество возможных шифртекстов равно 2^{128}, поэтому, с точки зрения противника, количество возможных открытых текстов тоже должно быть равно 2^{128}. Но если возможных ключей меньше, чем 2^{128}, то противник сможет исключить некоторые открытые тексты. Например, если длина ключа равна всего 64 бита, то противник сможет определить 2^{64} возможных открытых текстов и исключить подавляющее большинство 128-битовых строк. Противник не узнает, чему равен открытый текст, но узнает, чему он точно не равен, а уже этого достаточно, чтобы секретность шифрования была неидеальной.

Как видим, для обеспечения идеальной безопасности ключ должен быть не короче открытого текста, но в реальной жизни это требование быстро становится непрактичным. Далее мы обсудим современные подходы к шифрованию, позволяющие обеспечить оптимальную безопасность практически осуществимым способом.

Вероятность в криптографии

Вероятность количественно выражает шансы возникновения некоторого события. Это число от 0 до 1, причем 0 означает «никогда не произойдет», а 1 – «обязательно произойдет». Чем выше вероятность, тем больше шансов. В литературе можно найти различные объяснения вероятности, обычно речь идет о ящике с красными и белыми шарами и вероятности вытащить шар определенного цвета.

В криптографии вероятность часто используется для измерения шансов противника на успех. Для этого подсчитывается 1) число успешных событий (например, «найден единственно правильный секретный ключ») и 2) общее число возможных событий (например, общее число n-битовых ключей равно 2^n). В данном случае вероятность, что случайно выбранный ключ окажется правильным, равна $1/2^n$, т. е. отношению числа успешных событий (единственный секретный ключ) к общему числу событий (2^n возможных ключей). Число $1/2^n$ пренебрежимо мало для ключей типичной длины (128 и 256).

Вероятность того, что событие не *произойдет*, равна $1 - p$, где p – вероятность события. В примере выше вероятность получить неправильный ключ равна $1 - 1/2^n$, она очень близка к 1, можно считать, что почти наверняка так и будет.

Безопасность шифрования

Мы видели, что классические шифры небезопасны, а идеально безопасный шифр типа одноразового блокнота практически неудобен. Таким образом, мы должны несколько ослабить требования к безопасности, если хотим получить безопасные *и* практически применимые шифры. Но что вообще означает «безопасный», помимо очевидного и неформального «пассивный противник не может дешифрировать безопасные сообщения»?

Интуитивно представляется очевидным, что шифр безопасен, если даже при наличии большого числа пар «открытый текст – шифртекст» о его поведении *ничего нельзя узнать* при применении к другим открытым или шифртекстам. Это ставит новые вопросы.

- Как противник может заполучить такие пары? Насколько велико «большое число»? Все это определяется *моделями атак* – предположениями о том, что может и чего не может сделать противник.
- Что можно «узнать», и что такое «поведение шифра»? Это определяется *целями безопасности* (security goal) – описаниями того, что считать успешной атакой.

Модели атаки и цели безопасности неразделимы; нельзя заявить, что система безопасна, не объяснив, от кого и от чего ее защищают. Поэтому вводится понятие *аспекта безопасности* (security notion) – комбинации цели безопасности и модели атаки. Говорят, что шифр *реализует* некоторый аспект безопасности, если противник, работающий в рамках данной модели, не может достичь поставленной цели безопасности.

Модели атак

Модель атаки – это набор предположений о том, как противник мог бы взаимодействовать с шифром и что он может, а чего не может делать. Модель атаки должна:

- формулировать требования к криптографам, проектирующим шифры, чтобы они знали, что представляет собой противник, и от каких атак следует защищаться;
- давать пользователям указания о том, безопасно ли использовать шифр в конкретной ситуации;
- давать информацию криптоаналитикам, пытающимся взломать шифры, чтобы они понимали, допустима ли данная атака. Атака считается допустимой, только если она выполнима в контексте рассматриваемой модели.

Модели атак не обязаны в точности соответствовать реальности, это всего лишь аппроксимация. Статистик Джордж Э. П. Бокс писал: «Все модели неверны, но некоторые полезны». В криптографии, чтобы считаться полезной, модель атаки должна как минимум описы-

вать, какие действия доступны противнику для атаки на шифр. Нет ничего плохого в том, чтобы переоценить возможности противника, поскольку это помогает предвидеть будущие приемы атак – успеха добиваются лишь криптографы, страдающие паранойей. Плохая модель недооценивает противника и вселяет ложную уверенность в стойкости шифра, который кажется безопасным теоретически, но практически таковым не является.

Принцип Керкгоффса

Одно из предположений, присущее всем моделям, называется принципом Керкгоффса и утверждает, что безопасность шифра должна опираться только на секретность ключа, но не на секретность алгоритма. Сегодня, когда шифры и протоколы открыто публикуются и могут использоваться всеми желающими, это положение может показаться очевидным. Но в свое время голландский лингвист Огюст Керкгоффс говорил о военных шифровальных машинах, которые специально проектировались для некоторой армии или дивизии. В книге «Военная криптография», вышедшей в 1883 году, где были сформулированы шесть требований к военной криптографической системе, он писал: «Система не должна требовать секретности и при попадании в руки врага не должна терять надежности».

Модели черного ящика

Теперь рассмотрим некоторые полезные модели атак, формулируемые в терминах того, что противник может наблюдать и какие запросы он может предъявлять к шифру. *Запросом* в этом контексте называется операция, которая передает входное значение некоторой функции и получает в ответ результат; при этом детали работы функции не раскрываются.

Например, *запрос шифрования* принимает открытый текст и возвращает соответствующий шифртекст, не раскрывая секретного ключа.

Такие модели называются *моделями черного ящика*, потому что противник видит только данные на входе и выходе шифра. Например, некоторые микросхемы на смарт-картах безопасно защищают не только ключи шифра, но и его внутреннее устройство, но никто не мешает подключиться к микросхеме и попросить ее дешифрировать любой шифртекст. В результате противник получит соответствующий открытый текст, и это может помочь ему в определении ключа. Это реальный пример *запроса дешифрирования*.

Существует несколько разных моделей черного ящика. Я перечислю их в порядке расширения возможностей противника.

- *Атаки на основе шифртекста* (ciphertext-only attack – COA). Противник может наблюдать шифртекст, но не знает соответствующего ему открытого текста. Он также не знает, как выбирались открытые тексты. В модели COA противник пассивен и не может предъявлять запросов шифрования или дешифрирования.

- *Атаки с известным открытым текстом* (known-plaintext attack – KPA). Противник наблюдает шифртекст и знает соответствующий ему открытый текст. Таким образом, в модели KPA противник может получить список пар «открытый текст – шифртекст», но предполагается, что открытые тексты выбирались случайным образом. В этой модели KPA противник также пассивен.
- *Атаки с подобранным открытым текстом* (chosen-plaintext attack – CPA). Противник может предъявлять запросы шифрования по своему выбору и наблюдать результирующие шифртексты. Эта модель отражает ситуации, в которых противник имеет возможность задавать подлежащий шифрованию открытый текст полностью или частично и наблюдать соответствующий шифртекст. В отличие от пассивных моделей COA и KPA, в модели CPA противник *активен*, т. е. может оказывать воздействие на процессы шифрования, а не только пассивно подслушивать.
- *Атаки с подобранным шифртекстом* (chosen-ciphertext attack – CCA). Противник может предъявлять запросы шифрования и дешифрирования. На первый взгляд модель CCA может показаться абсурдной – раз мы можем дешифрировать, то что еще надо? Но, как и модель CPA, она представляет ситуации, когда противник может в какой-то мере повлиять на шифртекст, а впоследствии получить доступ к открытому тексту. Кроме того, возможность дешифрировать что-то не всегда означает вскрытие системы. Например, некоторые устройства защиты видео позволяют противнику предъявлять запросы шифрования и дешифрирования к микросхеме, но в этом контексте противник стремится получить ключ, что позволит ему самостоятельно распространять видео; способности «бесплатно» дешифрировать недостаточно для взлома системы.

В описанных выше моделях наблюдение и запросы шифртекстов обходятся не бесплатно. Каждый шифртекст является результатом вычисления функции шифрования. Это означает, что для генерирования 2^N пар «открытый текст – шифртекст» путем предъявления запросов шифрования требуется примерно столько же вычислений, сколько для проверки 2^N ключей. При оценивании стоимости атаки следует принимать во внимание стоимость выполнения запросов.

Модели серого ящика

В *модели серого ящика* противник имеет доступ к *реализации* шифра. Поэтому модели серого ящика более реалистичны, чем модели черного ящика в таких приложениях, как смарт-карты, встраиваемые и виртуализированные системы, к которым противник часто имеет физический доступ и потому может манипулировать внутренними аспектами алгоритмов. По той же причине модели серого ящика труднее определить, потому что они зависят от физических, аналоговых свойств, а не только от входов и выходов алгоритма, а теоретическая

криптография зачастую не способна абстрагировать всю сложность реального мира.

Одним из видов атак в рамках моделей серого ящика являются *атаки по побочным каналам*. Побочный канал – это источник информации, зависящий от реализации шифра, он может быть как программным, так и аппаратным. Противник, пользующийся побочным каналом, наблюдает или измеряет аналоговые характеристики реализации шифра, не нарушая его целостность; это *неразрушающие* атаки. Типичными примерами программных побочных каналов являются время выполнения и поведение системы, объемлющей шифр, например сообщения об ошибках, возвращаемые значения, ветвления и т. д. А для реализаций на смарт-картах типичная атака по побочному каналу включает измерение энергопотребления, электромагнитного излучения или акустического шума.

Разрушающие атаки на реализации шифров более эффективны, чем атаки по побочному каналу, но и стоят дороже, потому что требуют специального оборудования. Если для простой атаки по побочному каналу достаточно стандартного ПК и осциллографа, то для разрушающей атаки может потребоваться микроскоп с высокой разрешающей способностью и химическая лаборатория. Разрушающая атака включает целый набор приемов и процедур, от использования азотной кислоты для растворения корпуса микросхемы до получения микроскопических снимков, частичной обратной разработки и, возможно, модификации поведения микросхемы, например путем внесения неисправностей с помощью лазера.

Цели безопасности

Я неформально определил цель безопасности как «невозможность что-то узнать о поведении шифра». Чтобы превратить эту идею в строгое математическое определение, криптографы определяют две цели безопасности, соответствующие разным представлениям о том, что понимается под знанием о поведении шифра.

Неразличимость (indistinguishability – IND). Шифртексты должны быть неотличимы от случайных строк. Обычно это иллюстрируют на примере гипотетической игры: если противник выбирает два открытых текста, а затем получает шифртекст, соответствующий одному из них (выбранному наугад), то он не должен иметь возможность определить, какой открытый текст был зашифрован, даже если выполнит запросы шифрования для обоих открытых текстов (и запросы дешифрирования, если используется модель CCA, а не CPA).

Неподатливость (non-malleability – NM). Для заданного шифртекста $C_1 = \mathbf{E}(K, P_1)$ должно быть невозможно создать другой шифртекст C_2 такой, что соответствующий ему открытый текст P_2 каким-то осмысленным образом связан с P_1 (например, создать P_2, равный $P_1 \oplus 1$ или $P_1 \oplus X$ для некоторого известного значения X). Как ни

странно, одноразовый блокнот является податливым: если задан шифртекст $C_1 = P_1 \oplus K$, то можно определить $C_2 = C_1 \oplus 1$, являющийся допустимым шифртекстом для открытого текста $P_2 = P_1 \oplus 1$ с тем же ключом K. Вот тебе и идеальный шифр.

Далее мы обсудим эти цели безопасности в контексте различных моделей атак.

Аспекты безопасности

Цели безопасности полезны только в сочетании с моделью атаки. По соглашению, аспект безопасности записывают в виде *ЦЕЛЬ–МОДЕЛЬ*. Например, IND-CPA означает неразличимость для атак с подобранным открытым текстом, NM-CCA – неподатливость для атак с подобранным шифртекстом и т. д. Начнем с целей безопасности, представляющих интерес для противника.

Семантическая безопасность и рандомизированное шифрование: IND-CPA

Самым важным аспектом безопасности является IND-CPA, у него даже имеется специальное название *семантическая безопасность*. Он отражает интуитивную идею о том, что шифртекст не должен раскрывать никакой информации об открытом тексте, при условии что ключ секретный. Для достижения безопасности в смысле IND-CPA алгоритм шифрования должен возвращать разные шифртексты при неоднократном вызове для одного и того же открытого текста, иначе противник мог бы обнаружить повторяющиеся открытые тексты по их шифртекстам, что противоречит требованию о нераскрытии информации шифртекстом.

Один из способов добиться безопасности в смысле IND-CPA – воспользоваться *рандомизированным шифрованием*. Как следует из названия, в этом случае процесс шифрования рандомизируется и возвращает различные шифртексты при повторном шифровании одного и того же открытого текста. Тогда шифрование можно описать в виде $C = \mathbf{E}(K, R, P)$, где R – случайные биты, каждый раз выбираемые заново. Но дешифрирование остается детерминированным, потому что при заданном значении $\mathbf{D}(K, R, C)$ неизменно должно возвращать P, каким бы ни было значение R.

А что, если шифрование не рандомизировано? В игре IND, описанной в разделе «Цели безопасности» выше, противник выбирает два открытых текста, P_1 и P_2, и получает шифртекст для одного из них, но не знает, какого именно. То есть противник получает $C_i = \mathbf{E}(K, P_i)$ и должен угадать, чему равно i: 1 или 2. В модели CPA противник может выполнить запросы шифрования и найти $C_1 = \mathbf{E}(K, P_1)$ и $C_2 = \mathbf{E}(K, P_2)$. Если шифрование не рандомизировано, то достаточно посмотреть, чему равно $C_i - C_1$ или C_2, чтобы понять, какой открытый текст был зашифрован, и, стало быть, выиграть. Поэтому для аспекта безопасности IND-CPA рандомизация является неотъемлемой частью.

Примечание *При рандомизированном шифровании шифртекст может быть немного длиннее открытого текста, чтобы одному и тому же открытому тексту могло соответствовать несколько шифртекстов. Например, если одному открытому тексту может соответствовать 2^{64} шифртекстов, то шифртекст должен быть длиннее открытого по крайней мере на 64 бита.*

Достижение семантически безопасного шифрования

В одном из простейших построений семантически безопасного шифра используется *детерминированный генератор псевдослучайных битов* (deterministic random bit generator – DRBG) – алгоритм, который получает секретное значение и возвращает биты, выглядящие случайными:

$$\mathbf{E}(K, R, P) = (\mathbf{DRBG}(K||R) \oplus P, R).$$

Здесь строка R случайно выбирается для каждого нового акта шифрования и передается DRBG вместе с ключом ($K||R$ обозначает строку, полученную дописыванием R в конец K). Это напоминает одноразовый блокнот, только вместо выбора случайного ключа такой же длины, как у сообщения, мы пользуемся генератором псевдослучайных битов, чтобы получить строку, которая только выглядит как случайная.

Доказать, что этот шифр безопасен в смысле IND-CPA, легко, если предположить, что DRBG порождает случайные биты. Проведем доказательство от противного. Если мы можем различить шифртексты, полученные для разных случайных строк, т. е. можем отличить $\mathbf{DRBG}(K||R) \oplus P$ от случайного значения, значит, можем отличить и $\mathbf{DRBG}(K||R)$ от случайного значения. Напомним, что модель CPA позволяет получать шифртексты, соответствующие заданным открытым текстам P, поэтому мы можем применить XOR к P и $\mathbf{DRBG}(K||R) \oplus P$ и получить $\mathbf{DRBG}(K||R)$. Но это противоречит исходному предположению о том, что $\mathbf{DRBG}(K||R)$ нельзя отличить от случайного значения, т. е. порождаемые строки случайны. Итак, мы приходим к выводу, что шифртексты невозможно отличить от случайных строк, поэтому шифр безопасен.

Примечание *В качестве упражнения попробуйте установить, каким аспектам безопасности удовлетворяет шифр* $\mathbf{E}(K, R, P) = (\mathbf{DRBG}(K||R) \oplus P, R)$. *Безопасен ли он в смысле NM-CPA? А в смысле IND-CCA? Ответ будет дан в следующем разделе.*

Сравнение аспектов безопасности

Итак, мы узнали, что модели атак, например CPA и CCA, комбинируются с целями безопасности, допустим NM или IND, для построения аспектов безопасности NM-CPA, NM-CCA, IND-CPA и IND-CCA. Как соотносятся эти аспекты между собой? Можно ли доказать, что шифр, удовлетворяющий аспекту X, удовлетворяет также аспекту Y?

Некоторые соотношения очевидны: из IND-CCA следует IND-CPA, а из NM-CCA следует NM-CPA, потому что все действия, доступные CPA-противнику, доступны также CCA-противнику. То есть если невозможно взломать шифр, предъявляя запросы с подобранным шифртекстом и подобранным открытым текстом, то его невозможно взломать и с помощью одних лишь запросов с подобранным открытым текстом.

Не столь очевиден тот факт, что из IND-CPA не следует NM-CPA. Чтобы понять, почему это так, заметим, что показанное выше построение шифра IND-CPA (**DRBG**(K, R) $\oplus$ P, R) не обладает свойством NM-CPA: зная шифртекст (X, R), мы можем создать шифртекст ($X \oplus 1$, R), соответствующий $P \oplus 1$, что противоречит цели неподатливости.

Однако противоположное соотношение имеет место: из NM-CPA следует IND-CPA. Интуиция подсказывает, что система типа IND-CPA аналогична складыванию предметов в мешок: увидеть их нельзя, но можно перемешать, встряхнув мешок. NM-CPA больше похож на сейф: после того как предмет туда помещен, с ним уже никак нельзя взаимодействовать. Однако эта аналогия не работает для IND-CCA и NM-CCA, эквивалентным в том смысле, что из наличия одного аспекта следует наличие другого. Чисто техническое доказательство я опускаю.

Два типа приложений шифрования

Существует два основных типа приложений шифрования. *Транзитное шифрование* защищает данные, передаваемые с одной машины на другую: данные шифруются перед отправкой и дешифрируются после получения, как в случае зашифрованного соединения с сайтом электронной коммерции. *Стационарное шифрование* защищает данные, хранящиеся в информационной системе. Данные шифруются перед записью в память и дешифрируются перед чтением. Примерами могут служить системы шифрования диска на ноутбуках, а также шифрование виртуальных машин в облачных виртуальных экземплярах. Рассмотренные выше аспекты безопасности применимы к приложениям обоих типов, но какой аспект подходит лучше, может зависеть от приложения.

Асимметричное шифрование

До сих пор мы рассматривали только симметричное шифрование, когда обе стороны пользуются общим ключом. В случае *асимметричного шифрования* имеется два ключа: один для шифрования, другой для дешифрирования. Ключ шифрования называется *открытым*, и обычно считается, что он доступен любому, кто хочет отправить вам зашифрованное сообщение. Но ключ дешифрирования должен храниться в секрете, поэтому называется *закрытым*.

Открытый ключ можно вычислить по закрытому, но очевидно, что обратное невозможно. Иными словами, вычисление в одном направлении простое, а в другом трудное, в этом и заключается идея *криптографии с открытым ключом* – для функции, легко вычисляемой в одном направлении, практически невозможно вычислить обратную.

Модели атак и цели безопасности для асимметричного шифрования почти такие же, как для симметричного, но поскольку ключ шифрования открыт, любой противник может предъявлять запросы шифрования, используя этот ключ. Поэтому по умолчанию для асимметричного шифрования подразумевается модель атаки с подобранным открытым текстом (CPA).

Симметричное и асимметричное шифрования – два основных типа шифрования, именно они обычно применяются для построения безопасных систем связи. Они же используются как основа для более сложных схем, с которыми мы познакомимся ниже.

Дополнительные функции шифров

Базовый алгоритм шифрования преобразует открытые тексты в шифртексты и наоборот, не предъявляя никаких требований к безопасности. Но некоторым приложениям этого недостаточно, могут потребоваться дополнительные средства безопасности или дополнительная функциональность. Поэтому криптографы создали различные варианты симметричного и асимметричного шифрований. Одни из них хорошо известны, эффективны и широко распространены, другие пока являются экспериментальными, с ними трудно работать, а производительность не слишком высока.

Шифрование с аутентификацией

Шифрование с аутентификацией (authenticated encryption – AE) – вариант симметричного шифрования, при котором, помимо шифртекста, возвращается *аутентификационный жетон* (authentication tag). На рис. 1.4 показана схема шифрования с аутентификацией $\mathbf{AE}(K, P) = (C, T)$, где аутентификационный жетон T – короткая строка, которую невозможно угадать, не зная ключа. Алгоритм дешифрирования принимает K, C и T и возвращает открытый текст P, только если T является допустимым жетоном для этой пары «открытый текст – шифртекст»; в противном случае возвращается код ошибки.

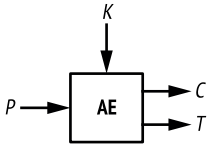

Рис. 1.4. Шифрование с аутентификацией

Жетон гарантирует *целостность* сообщения и служит доказательством того, что полученный шифртекст идентичен первоначально

отправленному правомочной стороной, знающей ключ *K*. Если *K* известен еще только одной стороне, то жетон гарантирует также, что сообщение было отправлено именно ей, т. е. он неявно *аутентифицирует* ожидаемого отправителя как фактического создателя сообщения.

Примечание *Я употребляю термин «создатель», а не «отправитель», потому что пассивный противник может подслушать и запомнить некоторые пары (C, T), отправленные стороной A стороне B, а затем снова отправить их B, притворившись A. Такую атаку повторным воспроизведением можно предотвратить, например, включив в сообщение счетчик. В момент дешифрирования сообщения его счетчик i увеличивается на 1. Таким образом, можно проверить, было ли сообщение отправлено дважды, и если да, то имеет место атака повторным воспроизведением. Одновременно это позволяет обнаружить потерянные сообщения.*

Шифрование с аутентификацией и ассоциированными данными (authenticated encryption with associated data – AEAD) – обобщение шифрования с аутентификацией, при котором некоторый открытый текст и незашифрованные данные используются для генерирования аутентификационного жетона **AEAD**(*K*, *P*, *A*) = (*C*, *T*). Типичное применение AEAD – защита дейтаграмм протоколов с открытым заголовком и зашифрованной полезной нагрузкой. В таких случаях по крайней мере какие-то данные заголовка должны оставаться открытыми; например, адреса получателей должны быть открыты, чтобы пакеты можно было маршрутизировать.

Дополнительные сведения о шифровании с аутентификацией см. в главе 8.

Шифрование с сохранением формата

Базовый шифр принимает и возвращает биты; ему безразлично, представляют ли биты текст, изображение или PDF-документ. Шифртекст тоже можно представить в виде неформатированных байтов, шестнадцатеричных символов, в кодировке base64 и в других форматах. Но что, если шифртекст должен иметь такой же формат, как открытый текст? Например, подобное требование предъявляется системами баз данных, которые могут хранить данные только в предписанном формате.

Эту задачу решает *шифрование с сохранением формата* (format-preserving encryption – FPE). Подобные алгоритмы создают шифртекст, имеющий такой же формат, как открытый текст. Например, FPE можно применить для шифрования IP-адресов (как показано на рис. 1.5), почтовых индексов, номеров кредитных карт с правильной контрольной суммой и т. д.

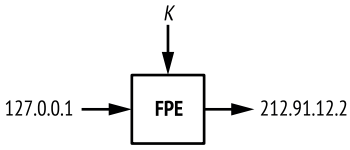

Рис. 1.5. Шифрование IP-адресов с сохранением формата

Полностью гомоморфное шифрование

Полностью гомоморфное шифрование (fully homomorphic encryption – FHE) – заветная мечта криптографа: оно позволяет пользователю заменить шифртекст $C = \mathbf{E}(K, P)$ другим шифртекстом $C' = \mathbf{E}(K, \mathbf{F}(P))$, где $\mathbf{F}(P)$ может быть произвольной функцией от P, не дешифрируя исходный шифртекст C. Например, P может быть текстовым документом, а **F** – модификацией части текста. Представьте себе облачное приложение, в котором хранятся ваши зашифрованные данные, но при этом облачный провайдер не знает ни характер данных, ни тип примененных к ним изменений. Звучит заманчиво, не правда ли?

Но у этой медали есть и обратная сторона: шифрование такого типа медленное – настолько медленное, что даже самые простые операции заняли бы недопустимо много времени. Первая схема FHE была придумана в 2009 году, с тех пор появились более эффективные варианты, но по-прежнему не ясно, сможет ли FHE когда-нибудь достичь быстроты, при которой станет полезным?

Шифрование, допускающее поиск

Такое шифрование позволяет искать по зашифрованной базе данных, не допуская утечки поисковых термов; с этой целью шифруется сам поисковый запрос. Как и полностью гомоморфное шифрование, шифрование с возможностью поиска могло бы повысить уровень конфиденциальности многих облачных приложений, скрыв поисковые запросы от облачного провайдера. Существуют коммерческие решения, рекламирующие возможность шифрования, допускающего поиск, но в большинстве случаев они основаны на стандартной криптографии, дополненной несколькими трюками, частично решающими проблему поиска. На момент написания этой книги шифрование, допускающее поиск, в исследовательском сообществе считается экспериментальной функцией.

Настраиваемое шифрование

Настраиваемое шифрование (tweakable encryption – TE) похоже на базовое, но в нем есть дополнительный настроечный параметр (tweak), призванный эмулировать различные версии шифра (см. рис. 1.6). Настроечный параметр может быть уникальным для каждого пользователя, это гарантирует, что пользовательский шифр не может быть клонирован другими сторонами, использующими тот же продукт. Основное применение TE – *шифрование диска*. Однако TE не привязано

к какому-то одному приложению, а является низкоуровневым типом шифрования и используется для построения других схем, например режимов шифрования в процессе аутентификации.

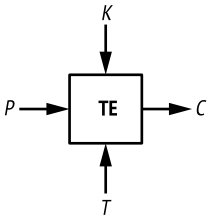

Рис. 1.6. Настраиваемое шифрование

При настраиваемом шифровании диска шифруется содержимое запоминающих устройств, например жестких или SSD-дисков. (Применять здесь рандомизированное шифрование нельзя, потому что при этом увеличился бы размер данных, что для файлов на носителе недопустимо.) Чтобы шифрование было непредсказуемым, используется настроечный параметр, зависящий от положения шифруемых данных, обычно номер сектора или индекс блока.

Какие возможны проблемы

Алгоритмы шифрования или их реализации могут не обеспечивать конфиденциальность по многим причинам. Например, могут не удовлетворяться требования к безопасности («система должна быть безопасной в смысле IND-CPA»), или сформулированные требования могут не отвечать реальности (скажем, в технических условиях прописана только безопасность в смысле IND-CPA, тогда как противник может предъявлять запросы с подобранным шифртекстом). Увы, многие инженеры даже не задумываются о требованиях к криптографической безопасности, а просто хотят, чтобы система была «безопасной», не понимая, что это означает. Это прямой путь к катастрофе. Рассмотрим два примера.

Слабый шифр

Наш первый пример касается шифров, которые можно атаковать методами криптоанализа, как случилось со стандартом мобильной связи 2G. В мобильных телефонах 2G использовался шифр A5/1, оказавшийся слабее, чем ожидалось, поэтому любой человек, обладающий соответствующими навыками и инструментами, мог перехватывать вызовы. Операторы связи вынуждены были искать обходные пути для предотвращения атак.

Примечание *В стандарте 2G был определен также шифр A5/2 для всех регионов, кроме ЕС и США. Он преднамеренно был сделан более слабым, чтобы помешать повсеместному применению стойкого шифрования.*

Заметим, впрочем, что атака на шифр A5/1 нетривиальна, исследователям потребовалось более 10 лет для открытия эффективного криптоаналитического метода. Кроме того, для атаки требуется *компромисс между временем и памятью* (time-memory trade-off – TMTO), т. е. сначала нужно потратить несколько дней или недель на вычисление очень больших справочных таблиц, которые затем применяются для проведения самой атаки. В случае A5/1 заранее вычисленные таблицы занимают более 1 ТБ. В последующих стандартах мобильного шифрования, в частности 3G и LTE, определены более стойкие шифры, но это не значит, что система в целом не может быть скомпрометирована; просто шифрование нельзя скомпрометировать, взломав симметричный шифр, являющийся частью системы.

Неправильная модель

В следующем примере рассматривается неправильная модель атаки, не учитывающая некоторых побочных каналов.

Во многих коммуникационных протоколах с шифрованием гарантируется, что используются шифры, которые считаются безопасными в модели CPA или CCA. Однако для некоторых атак не нужны запросы шифрования, как в модели CPA, или не используются запросы дешифрирования, как в модели CCA. Требуются только *запросы допустимости* – является ли данный шифртекст допустимым, и эти запросы обычно отправляются системе, отвечающей за дешифрирование шифртекстов. Примером могут служить *атаки на оракул дополнения* (padding oracle attack), когда противник узнает, правильно ли отформатирован шифртекст.

Точнее, в случае атаки на оракул дополнения шифртекст является допустимым, только если соответствующий ему открытый текст правильно *дополнен* последовательностью байтов с целью упростить шифрование. Если дополнение неправильно, то дешифрирование завершится ошибкой, а противник часто может обнаружить ошибку дешифрирования и попытаться эксплуатировать ее. Например, в программе на Java исключение `javax.crypto.BadPaddingException` означает, что открытый текст был дополнен неправильно.

В 2010 году исследователи нашли возможности для атаки на оракул дополнения в нескольких серверах веб-приложений. Запрос допустимости заключался в отправке шифртекста некоторой системе, после чего анализировалось, была ли ошибка. С помощью таких запросов удалось дешифрировать безопасные в остальных отношениях шифртексты, не зная ключа.

Криптографы часто игнорируют атаки на оракул дополнения, потому что обычно они зависят от поведения приложения и от способа взаимодействия с ним пользователей. Но если вы не предвидите таких атак и не включаете их в модель при проектировании и развертывании криптографической системы, то можете столкнуться с неприятными сюрпризами.

Для дополнительного чтения

В этой книге мы будем подробно обсуждать шифрование и его различные формы, особенно работу современных безопасных шифров. Но не сможем охватить все на свете, так что многие увлекательные темы останутся за бортом. Например, чтобы изучить теоретические основания шифрования и глубже познакомиться с понятием неразличимости (IND), рекомендую прочитать вышедшую в 1982 году статью Goldwasser and Micali «Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information», в которой впервые была сформулирована идея семантической безопасности. Если вас интересуют физические атаки и криптографическое оборудование, то обратитесь к материалам конференции CHES (Conference on Cryptographic Hardware and Embedded Systems).

Существует гораздо больше типов шифрования, чем упомянуто в этой главе, в т. ч. шифрование на основе атрибутов, широковещательное шифрование, функциональное шифрование, личностное шифрование, шифрование с фиксацией сообщения, перешифрование с посредником – и это далеко не всё. Последние работы на эту тему можно найти на сайте *https://eprint.iacr.org/*, где хранится электронный архив научных статей по криптографии.

2

СЛУЧАЙНОСТЬ

Случайность встречается в криптографии повсеместно: при генерировании секретных ключей, в схемах шифрования и даже в атаках на криптосистемы. Без случайности не было бы никакой криптографии, потому что все операции были бы предсказуемы, а значит, небезопасны.

Эта глава является введением в понятие случайности в контексте криптографии и ее приложений. Мы обсудим генераторы псевдослучайных чисел и то, как операционные системы могут дать доступ к надежным источникам истинной случайности, а завершим главу реальными примерами, показывающими, как дефектная реализация случайности может повлиять на безопасность.

Случайное или неслучайное?

Вы наверняка слышали выражение «случайные биты», но, строго говоря, нет такой вещи, как последовательность случайных битов. На самом деле речь идет об алгоритме или процессе, порождающем по-

следовательность битов, поэтому, говоря «случайные биты», мы имеем в виду случайно сгенерированные биты.

И как же выглядят случайные биты? Большинству людей последовательность 8 бит 11010110 покажется более случайной, чем 00000000, хотя вероятность генерирования той и другой одинакова (и равна 1/256). Значение 11010110 выглядит более случайным, чем 00000000, потому что обладает признаками, типичными для случайно сгенерированного значения, т. е. в нем нет очевидной закономерности.

Видя строку 11010110, наш мозг отмечает, что в ней примерно столько же нулей (три), сколько единиц (пять), как еще в 55 восьмибитовых строках (11111000, 11110100, 11110010 и т. д.). А 8-битовая строка, состоящая только из нулей, всего одна. Поскольку паттерн, содержащий три нуля и пять единиц, встречается чаще, чем паттерн, содержащий восемь нулей, мы считаем, что последовательность 11010110 случайная, а 00000000 неслучайная. И если программа породит последовательность 11010110, то вы подумаете, что она случайная, даже если на самом деле это не так. Напротив, если рандомизированная программа породит последовательность 00000000, то вы, наверное, усомнитесь в ее случайности.

Этот пример иллюстрирует два типа ошибок, часто допускаемых людьми при распознавании случайности:

- **принятие неслучайного за случайное.** Думают, что объект был сгенерирован случайно, только потому, что он *выглядит* случайным;
- **принятие случайного за неслучайное.** Думают, что паттерн, возникший случайно, на самом деле имеет какую-то скрытую причину.

Различие между тем, что выглядит случайным, и тем, что является случайным на самом деле, очень важно. Более того, в криптографии неслучайность часто оказывается синонимом небезопасности.

Случайность как распределение вероятностей

Любой случайный процесс можно охарактеризовать *распределением вероятностей*, которое дает все, что нужно знать о случайности процесса. Распределение вероятностей, или просто распределение, перечисляет исходы случайного процесса и сопоставляет каждому исходу его *вероятность*.

Вероятность измеряет шансы возникновения события. Она выражается вещественным числом от 0 до 1, причем вероятность 0 означает, что событие невозможно, а вероятность 1 – что оно произойдет наверняка. Например, при подбрасывании монеты каждая сторона выпадает с вероятностью 1/2, а вероятность, что монета встанет на ребро, обычно считается нулевой.

Распределение вероятностей должно включать все возможные исходы, так чтобы сумма вероятностей была равна 1. Формально, если

существует N возможных исходов, то распределение содержит N вероятностей $p_1, p_2, \ldots, p_N$, причем $p_1 + p_2 + \ldots + p_N = 1$. В случае подбрасывания монеты распределение содержит два исхода: выпадение орла и выпадение решки, вероятность каждого из которых равна 1/2. Сумма обеих вероятностей 1/2 + 1/2 = 1, поскольку монета обязательно ляжет на одну сторону.

При *равномерном распределении* вероятности всех исходов одинаковы. Если имеется N событий, то вероятность каждого равна $1/N$. Например, если 128-битовый ключ случайно выбирается из равномерного распределения, то каждый из 2^{128} возможных ключей будет иметь вероятность $1/2^{128}$.

С другой стороны, если распределение *неравномерно*, то вероятности не равны. Если результаты подбрасывания монеты имеют неравномерное распределение, то монета называется несимметричной; например, орел может выпадать с вероятностью ¼, а решка – с вероятностью ¾.

Энтропия: мера неопределенности

Энтропия измеряет неопределенность, или беспорядочность, системы. Можно считать, что энтропия описывает удивление, которое мы испытываем, глядя на результат случайного процесса: чем выше энтропия, тем меньше уверенность в результате.

Энтропию распределения вероятностей можно вычислить. Если распределение состоит из вероятностей $p_1, p_2, \ldots, p_N$, то его энтропия равна сумме произведений вероятностей на их логарифмы, взятой с обратным знаком:

$$-p_1 \times \log(p_1) - p_2 \times \log(p_2) \ldots - p_N \times \log(p_N).$$

Здесь под *log* понимается *двоичный логарифм*, т. е. логарифм по основанию 2. В отличие от натурального, двоичный логарифм выражает величину информации в битах и дает целые значения для вероятностей, являющихся степенями двойки. Например, $\log(1/2) = -1$, $\log(1/4) = -2$ и вообще $\log(1/2^n) = -n$. (Именно поэтому сумма берется с обратным знаком, чтобы в результате получилось положительное число.) Таким образом, для случайных равномерно распределенных 128-битовых ключей энтропия равна:

$$2^{128} \times (-2^{-128} \times \log(2^{-128})) = -\log(2^{-128}) = 128 \text{ бит}.$$

Если заменить 128 произвольным целым числом n, то окажется, что энтропия равномерно распределенных n-битовых строк равна n бит.

Энтропия максимальна, когда распределение равномерно, потому что при таком распределении максимальна неопределенность: ни один исход не является более вероятным, чем любой другой. Поэтому n-битовые значения не могут иметь более n бит энтропии.

Рассуждая таким же образом, приходим к выводу, что если распределение неравномерно, то энтропия меньше. Рассмотрим пример с подбрасыванием монеты. Для симметричной монеты энтропия равна

$$-(1/2) \times \log(1/2) - (1/2) \times \log(1/2) = 1/2 + 1 + 2 = 1 \text{ бит.}$$

А что, если вероятность выпадения одной стороны больше, чем другой? Скажем, орел выпадает с вероятностью ¼, а решка с вероятностью 3/4 (напомним, что сумма вероятностей должна быть равна 1).

Энтропия такого несимметричного подбрасывания равна

$$-(3/4) \times \log(3/4) - (1/4) \times \log(1/4) \approx -(3/4) \times (-0.415) - (1/4) \times (-2) \approx 0.81 \text{ бита.}$$

Из того, что 0.81 меньше 1 (энтропия подбрасывания симметричной монеты), следует, что чем менее симметрична монета, тем менее равномерно распределение и тем меньше энтропия. Если орел выпадает с вероятностью 1/10, то энтропия будет равна 0.469, а если с вероятностью 1/100, то энтропия упадет до 0.081.

Примечание *Энтропию можно интерпретировать также как меру информации. Например, результат подбрасывания симметричной монеты дает ровно один бит информации – орел или решка, и заранее предсказать результат подбрасывания невозможно. В случае подбрасывания несимметричной монеты мы заранее знаем, что выпадение орла более вероятно, поэтому можем предсказать исход подбрасывания. Подбрасывание монеты дает информацию, необходимую для предсказания результата с некоторой уверенностью.*

Генераторы случайных и псевдослучайных чисел

Для безопасности криптосистем случайность зачастую необходима, поэтому нам нужен компонент, от которого можно получать случайные данные. Задача этого компонента – возвращать по запросу случайные биты. Как же генерируется случайность? Нам нужны две вещи:

- источник неопределенности, или *источник энтропии*, – это генератор случайных чисел (random number generator – RNG);
- криптографический алгоритм, порождающий случайные биты высокого качества по этому источнику энтропии. Такой алгоритм дают генераторы псевдослучайных чисел (pseudorandom number generator – PRNG).

Использование RNG и PRNG – ключ к тому, чтобы сделать криптографию практичной и безопасной. Прежде чем переходить к изучению PRNG, рассмотрим вкратце, как работают RNG.

Случайность присуща окружающей среде, которая по природе своей аналоговая, хаотичная, неопределенная и потому непредсказуемая. Сгенерировать случайность, опираясь только на компьютерные алго-

ритмы, невозможно. В криптографии случайные данные обычно получают от *генераторов случайных чисел* (RNG) – программных или аппаратных компонентов, которые задействуют случайность аналоговых элементов для порождения непредсказуемых битов в цифровой системе. Например, RNG может напрямую получать биты в результате измерений температуры, акустического шума, турбулентности воздуха или статического электричества. К сожалению, такие аналоговые источники энтропии не всегда доступны, а их энтропию трудно оценить.

RNG могут также получать энтропию от работающей операционной системы: присоединенных к ней датчиков, устройств ввода-вывода, активности сети или диска, системных журналов, работающих процессов и действий пользователей, например нажатия клавиш и перемещения мыши. Такие события, генерируемые системой и человеком, могут оказаться хорошим источником энтропии, но они ненадежны и могут контролироваться противником. Кроме того, они слишком медленны для получения случайных битов.

Квантовые генераторы случайных чисел (quantum random number generator – QRNG) опираются на случайность квантово-механических явлений: радиоактивного распада, флуктуаций в вакууме и наблюдения поляризации фотонов. Они могут дать настоящую, а не кажущуюся случайность. Но на практике QRNG могут иметь статистическое смещение и порождают биты недостаточно быстро; как и вышеупомянутые источники энтропии, они нуждаются в дополнительном компоненте, чтобы возвращать случайные биты с высокой скоростью.

Генераторы псевдослучайных чисел (PRNG) решают проблему, с которой мы столкнулись, порождая много искусственно случайных битов по немногим истинно случайным. Например, RNG, транслирующий перемещения мыши в случайные биты, перестал бы работать, если бы мы прекратили трогать мышь. А PRNG неизменно возвращает псевдослучайные биты в ответ на запрос.

PRNG опираются на RNG, но ведут себя иначе: RNG недетерминированно порождают истинно случайные биты из аналоговых источников, но медленно и без гарантии высокой энтропии. С другой стороны, PRNG детерминированно порождают кажущиеся случайными биты с высокой энтропией из цифровых источников. По сути дела, PRNG трансформирует немногие ненадежные случайные биты в длинный поток надежных псевдослучайных битов, пригодный для криптографических приложений (см. рис. 2.1).

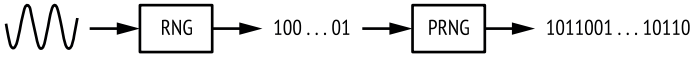

Рис. 2.1. RNG порождает немного ненадежных битов из аналоговых источников, а PRNG расширяют эти биты в длинный поток надежных битов

Как работает PRNG

PRNG получает случайные биты от RNG с регулярными интервалами и использует их для обновления большого буфера памяти, который

называется *пулом энтропии*. Пул энтропии – это источник энтропии для PRNG, точно так же, как физическая среда является таковым для RNG. Обновляя пул энтропии, PRNG смешивает хранящиеся в нем биты между собой, чтобы избавиться от статистического смещения.

Для генерирования псевдослучайных битов PRNG выполняет алгоритм детерминированного генератора случайных битов (DRBG), который порождает из битов, находящихся в пуле энтропии, гораздо более длинную последовательность. Как следует из самого названия, алгоритм DRBG детерминированный, а не рандомизированный: получив одинаковые данные на входе, он возвращает одинаковые результаты на выходе. PRNG гарантирует, что его DRBG никогда не получит одинаковые входные данные дважды; это необходимо, чтобы генерируемые псевдослучайные последовательности были уникальны.

В процессе работы PRNG выполняет три операции:

- ***init()*** инициализирует пул энтропии и внутреннее состояние PRNG;
- ***refresh(R)*** обновляет пул энтропии, используя некоторые данные *R*, обычно получаемые от RNG;
- ***next(N)*** возвращает *N* псевдослучайных бит и обновляет пул энтропии.

Операция *init* сбрасывает PRNG в начальное состояние: заново инициализирует пул энтропии значением по умолчанию, а переменные и буферы памяти подготавливает к выполнению операций *refresh* и *next*.

Операцию *refresh* часто называют *доинициализацией* (reseeding), а ее аргумент *R – начальным значением* (seed). Если RNG недоступен, то в качестве начальных должны использоваться уникальные значения, зашитые в код системы. Обычно операцию *refresh* вызывает операционная система, а операцию *next* – приложения. Операция *next* выполняет DRBG и модифицирует пул энтропии, так чтобы следующий вызов гарантированно дал другие псевдослучайные биты.

Вопросы безопасности

Теперь поговорим немного о том, как PRNG решает некоторые высокоуровневые проблемы безопасности. Конкретно PRNG должен гарантировать *устойчивость к ретроанализу* (backtracking resistance) и *устойчивость к предсказанию* (prediction resistance). Устойчивость к ретроанализу (ее еще называют *секретностью прошлого*) означает, что невозможно восстановить ранее сгенерированные биты, а устойчивость к предсказанию (*обратная секретность*) – что невозможно предсказать будущие биты.

Для обеспечения устойчивости к ретроанализу PRNG должен позаботиться о том, чтобы преобразования, выполняемые при обновлении состояния операциями *refresh* и *next*, были необратимы. Тогда, даже если противник скомпрометирует систему и узнает состояние

пула энтропии, он не сможет определить предыдущее состояние пула и ранее сгенерированные биты. Для обеспечения устойчивости к предсказанию PRNG должен регулярно выполнять операцию *refresh*, задавая значения *R*, неизвестные противнику и не поддающиеся угадыванию. Это помешает противнику определить будущие значения пула энтропии, даже если весь пул будет скомпрометирован. (Даже если список использованных значений *R* известен, для реконструкции пула нужно знать, в каком порядке вызывались операции *refresh* и *next*.)

PRNG Fortuna

PRNG-генератор *Fortuna*, используемый в Windows, был разработан в 2003 году Нильсом Фергюсоном и Брюсом Шнейером. Этот алгоритм заменил систему *Yarrow*, которая была разработана в 1998 году Келси и Шнейнером и по сей день используется в операционных системах macOS и iOS. Я не стану здесь приводить спецификацию Fortuna и описывать его реализацию, но попытаюсь объяснить, как он работает. Полное описание Fortuna можно найти в главе 9 книги Ferguson, Schneier, Kohno «Cryptography Engineering» (Wiley, 2010).

Внутренняя память Fortuna включает следующие компоненты:

- 32 пула энтропии $P_1, P_2, \ldots, P_{32}$, организованных так, что P_i используется в каждой 2^i-й доинициализации;
- ключ *K* и счетчик *C* (оба длиной 16 байт). Они образуют внутреннее состояние DRBG, применяемого в Fortuna.

Упрощенно работу Fortuna можно описать следующим образом:

- *init*() инициализирует *K* и *C* нулями и опустошает все 32 пула энтропии P_i;
- *refresh*(*R*) добавляет данные *R* в один из пулов энтропии. Система выбирает, какие RNG использовать для порождения значений *R*, и должна регулярно вызывать *refresh*;
- *next*(*N*) обновляет *K*, используя один или несколько пулов энтропии, причем выбор пула определяется в первую очередь тем, сколько обновлений *K* уже было произведено. Затем *N* запрошенных бит порождаются путем шифрования *C* на ключе *K*. Если шифрования *C* недостаточно, то Fortuna шифрует *C* + 1, потом *C* + 2 и т. д., пока не наберет достаточно битов.

Хотя работа алгоритма Fortuna выглядит довольно просто, правильно реализовать его нелегко. Во-первых, нужно тщательно продумать все детали алгоритма: как выбираются пулы энтропии, какой тип шифра использовать в *next*, что делать, если не получено никакой энтропии, и т. д. В спецификации описано большинство деталей, но не включен полный набор тестов для проверки правильности реализации, поэтому трудно убедиться, что реализация Fortuna ведет себя, как задумано.

Но даже если Fortuna реализован корректно, возможны ошибки безопасности по причинам, не связанным с правильностью реализации. Например, Fortuna может не заметить, что RNG дает недостаточно случайных битов, и в результате будет порождать псевдослучайные биты низкого качества или вообще прекратит отдавать псевдослучайные биты.

Еще один риск, присущий реализациям Fortuna, – возможность раскрытия *файлов начальных значений* противнику. Данные, хранящиеся в файлах начальных значений, используются для снабжения вызовов *refresh* энтропией в тех случаях, когда RNG недоступен, например сразу после перезагрузки системы, еще до того, как системные RNG зарегистрировали какие-то непредсказуемые события. Но если один и тот же файл начального значения будет использован дважды, то Fortuna дважды сгенерирует одну и ту же последовательность битов. Поэтому файл начального значения следует стирать после использования, чтобы предотвратить его повторное использование.

Наконец, если два экземпляра Fortuna находятся в одном и том же состоянии, потому что разделяют общий файл начального значения (а значит, в пулах энтропии находятся одинаковые данные и *C* и *K* тоже одинаковы), то операция *next* обоих экземпляров будет возвращать одни и те же биты.

Криптографически стойкие и нестойкие PRNG

PRNG бывают криптографически стойкими и нестойкими. Криптографически нестойкие PRNG предназначены для порождения равномерных распределений в таких приложениях, как естественно-научное моделирование или видеоигры. Однако их ни в коем случае не следует использовать в криптографических приложениях, поскольку они небезопасны, их основная задача – обеспечить качественное распределение вероятностей битов, а не их непредсказуемость. С другой стороны, криптографически стойкие PRNG непредсказуемы, поскольку в них уделяется внимание также стойкости элементарных *операций*, применяемых для порождения битов с качественным распределением.

К сожалению, большинство PRNG, доступных из языков программирования, в частности `rand` и `drand48` в libc, `rand` и `mt_rand` в PHP, модуль Python `random`, класс Ruby `Random` и т. д., являются криптографически нестойкими. Использовать криптографически нестойкий PRNG по умолчанию – прямой путь к беде, потому что часто он просачивается и в криптографические приложения. Поэтому следите за тем, чтобы в криптографических приложениях использовались только криптографически стойкие PRNG.

Популярный криптографически нестойкий PRNG: вихрь Мерсенна

Вихрь Мерсенна (Mersenne Twister – MT) – это криптографически нестойкий алгоритм PRNG, используемый (на момент написания кни-

ги) в PHP, Python, R, Ruby и многих других системах. MT генерирует равномерно распределенные случайные биты, статистически несмещенные, но предсказуемые: зная несколько битов, порожденных MT, не слишком сложно предсказать следующие.

Рассмотрим, что именно делает вихрь Мерсенна небезопасным. Алгоритм MT гораздо проще криптографически стойких PRNG: в качестве внутреннего состояния используется массив S, состоящий из 624 32-битовых слов. Первоначально массив содержит элементы S_1, S_2, ..., S_{624}, затем S_2, ..., S_{625}, далее S_3, ..., S_{626} и т. д., вычисляемые по формуле

$$S_{k+624} = S_{k+397} \oplus \mathbf{A}((S_k \wedge 0x80000000) \vee (S_{k+1} \wedge 0x7fffffff)).$$

Здесь $\oplus$ обозначает поразрядную операцию XOR (^ в языке программирования C), $\wedge$ – поразрядное AND (& в C), $\vee$ – поразрядное OR (| в C), а **A** – функция, преобразующая 32-битовое слово x в (x >> 1), если старший бит x равен 0, или в (x >> 1) $\oplus$ 0x9908b0df в противном случае.

Обратите внимание, что в этой формуле биты S взаимодействуют друг с другом только посредством операций XOR. Операторы $\wedge$ и $\vee$ никогда не объединяют два бита S, а только биты S с константами 0x80000000 и 0x7fffffff. Таким образом, любой бит S_{625} можно выразить как результат применения XOR к битам S_{398}, S_1 и S_2, и вообще любой бит будущего состояния – в виде результата применения XOR к битам начального состояния S_1, ..., S_{624}. (В выражении $S_{228+624} = S_{852}$ в виде функции от S_{625}, S_{228} и S_{229} можно заменить S_{625} его выражением через S_{398}, S_1 и S_2.)

Поскольку в начальном состоянии имеется ровно $624 \times 32 = 19\,968$ бит (или 624 32-битовых слова), то любой выходной бит можно выразить в виде формулы, содержащей не более 19 969 членов (19 968 бит плюс один постоянный бит). Это примерно 2,5 килобайта данных. Обратное также верно: биты начального состояния можно выразить в виде результата применения XOR к выходным битам.

Небезопасность линейности

Мы называем XOR-комбинацию битов *линейной комбинацией*. Например, если X, Y и Z – биты, то выражение $X \oplus Y \oplus Z$ является линейной комбинацией, а $(X \wedge Y) \oplus Z$ не является из-за присутствия оператора AND ($\wedge$). Если изменить значение бита X на противоположное, то результат вычисления $X \oplus Y \oplus Z$ тоже изменится вне зависимости от значений Y и Z. С другой стороны, если изменить на противоположное значение бита X в выражении $(X \wedge Y) \oplus Z$, то результат изменится, лишь если бит Y в той же позиции равен 1. Таким образом, линейные комбинации предсказуемы, поскольку необязательно знать значения битов, чтобы предсказать, как их изменение отразится на результате.

Если бы алгоритм MT был криптографически стойким, то его формулы были бы *нелинейными* и включали бы не только отдельные биты,

но также AND-комбинации (*произведения*) битов, например $S_1S_{15}S_{182}$ или $S_{17}S_{256}S_{257}S_{354}S_{498}S_{601}$. Линейные комбинации этих битов включают не более 624 переменных, тогда как нелинейные допускали бы до 2^{624} переменных. Было бы невозможно не только решить, но хотя бы записать эти уравнения. (Заметим, что гораздо меньшим числом, 2^{305}, оценивается количество информации в наблюдаемой Вселенной.)

Суть дела в том, что линейные преобразования ведут к более коротким уравнениям (размер которых сопоставим с числом переменных), которые проще решать, тогда как нелинейные приводят к уравнениям экспоненциального размера, практически неразрешимым. Поэтому цель криптографов – проектирование алгоритмов PRNG, которые эмулировали бы такие сложные нелинейные преобразования с помощью небольшого числа простых операций.

Примечание *Линейность – лишь один из многих критериев безопасности. Хотя нелинейность является необходимым условием, ее одной еще недостаточно, чтобы сделать PRNG криптографически стойким.*

Полезность статистических тестов

Пакеты статистических тестов, в частности TestU01, Diehard и пакет Национального института стандартов и технологий (NIST), – один из способов проверить качество псевдослучайных битов. Такие тесты принимают выборку псевдослучайных битов, сгенерированную PRNG (скажем, объемом 1 мегабайт), вычисляют некоторые статистики распределения определенных паттернов и сравнивают результаты с полученными для идеального равномерного распределения. Например, некоторые тесты подсчитывают количество единиц и нулей или распределение 8-битовых паттернов. Но статистические тесты никак не связаны с криптографической безопасностью, можно спроектировать криптографически нестойкий PRNG, который пройдет любой статистический тест.

Результатом прогона статистических тестов для случайно сгенерированных данных обычно является набор статистических показателей. Как правило, это *P*-значения, широко распространенный статистический индикатор. Такие результаты бывает нелегко интерпретировать, потому что это не просто «прошел» или «не прошел». Если полученные при первом прогоне результаты выглядят аномально, не переживайте: возможно, причиной является случайное возмущение или тестирование проводилось на слишком малой выборке. Чтобы убедиться в нормальности результатов, сравните их с результатами, полученными для какой-то надежной выборки такого же размера, например сгенерированной следующей командой из комплекта программ OpenSSL:

```
$ openssl rand <число байтов> -out <выходной файл>
```

PRNG на практике

Теперь поговорим о практической реализации PRNG. Криптографически стойкие PRNG входят в состав операционных систем (ОС) на большинстве платформ, от персональных компьютеров и ноутбуков до встраиваемых систем, например маршрутизаторов и ТВ-приставок, а также имеются в виртуальных машинах, мобильных телефонах и т. д. Большая часть этих PRNG реализована программно, но встречаются и чисто аппаратные. Такие PRNG используются приложениями, исполняемыми под управлением ОС, а иногда другими PRNG, работающими поверх криптографических библиотек или приложений.

Далее мы рассмотрим самые распространенные PRNG: один работает в Linux, Android и многих других системах на базе Unix, другой – в Windows, а третий, реализованный аппаратно, – в недавних версиях микропроцессоров Intel.

Генерирование случайных битов в системах на базе Unix

Устройство */dev/urandom* – работающий в пространстве пользователя интерфейс к криптографически стойкому PRNG в распространенных системах *nix, именно он обычно используется для генерирования надежных случайных битов. Поскольку это устройство, для запроса битов нужно читать его как файл. Например, в следующей команде */dev/urandom* используется для записи 10 МБ случайных бит в файл:

```
$ dd if=/dev/urandom of=<выходной файл> bs=1M count=10
```

Неправильное использование /dev/urandom

Можно было бы написать на C наивную и небезопасную программу чтения случайных битов типа показанной в листинге 2.1 и надеяться на авось, но это дурная идея.

Листинг 2.1. Небезопасное использование `/dev/urandom`

```
int random_bytes_insecure(void *buf, size_t len)
{
    int fd = open("/dev/urandom", O_RDONLY);
    read(fd, buf, len);
    close(fd);
    return 0;
}
```

Этот код небезопасен; он даже не проверяет значения, возвращаемые функциями `open()` и `read()`, а это означает, что буфер, в котором вы ожидаете увидеть случайные биты, может оказаться заполненным нулями или вообще не изменится.

Более безопасный способ работы с /dev/urandom

В листинге 2.2, скопированном из библиотеки LibreSSL, показано более безопасное использование */dev/urandom*.

Листинг 2.2. Безопасное использование /dev/urandom

```
int random_bytes_safer(void *buf, size_t len)
{
    struct stat st;
    size_t i;
    int fd, cnt, flags;
    int save_errno = errno;

start:
    flags = O_RDONLY;
#ifdef O_NOFOLLOW
    flags |= O_NOFOLLOW;
#endif
#ifdef O_CLOEXEC
    flags |= O_CLOEXEC;
#endif
    fd = ❶open("/dev/urandom", flags, 0);
    if (fd == -1) {
        if (errno == EINTR)
            goto start;
        goto nodevrandom;
    }
#ifndef O_CLOEXEC
    fcntl(fd, F_SETFD, fcntl(fd, F_GETFD) | FD_CLOEXEC);
#endif

    /* Простая проверка разумного поведения устройства */
    if (fstat(fd, &st) == -1||!S_ISCHR(st.st_mode)) {
        close(fd);
        goto nodevrandom;
    }
    if (ioctl(fd, RNDGETENTCNT, &cnt) == -1) {
        close(fd);
        goto nodevrandom;
    }
    for (i = 0; i < len; ) {
        size_t wanted = len - i;
        ssize_t ret = ❷read(fd, (char *)buf + i, wanted);

        if (ret == -1) {
            if (errno == EAGAIN||errno == EINTR)
                continue;
            close(fd);
            goto nodevrandom;
        }
        i += ret;
    }
```

```
    close(fd);
    if (gotdata(buf, len) == 0) {
        errno = save_errno;
        return 0;    /* все хорошо */
    }
nodevrandom:
    errno = EIO;
    return -1;
}
```

В отличие от листинга 2.1, в листинге 2.2 производятся некоторые проверки. Сравните, например, вызовы `open()` в точке ❶ и `read()` в точке ❷ с аналогичными вызовами в листинге 2.1: в безопасной версии проверяются значения, возвращенные функциями, и в случае ошибки файловый дескриптор закрывается и возвращается –1.

Различия между /dev/urandom и /dev/random в Linux

В различных версиях Unix используются разные PRNG. В Linux PRNG, код которого находится в файле *drivers/char/random.c*, вызывает функцию хеширования SHA-1 для преобразования исходных битов энтропии в надежные псевдослучайные биты. PRNG получает энтропию из различных источников (включая клавиатуру, мышь, диск и моменты прерываний) и имеет главный пул энтропии размером 512 байт, а также неблокирующий пул для */dev/urandom* и блокирующий пул для */dev/random*.

В чем разница между */dev/urandom* и */dev/random*? Вкратце: */dev/random* пытается оценить количество энтропии и отказывается возвращать биты, если ее уровень слишком низок. На первый взгляд, идея здравая, но на самом деле таковой не является. Во-первых, алгоритмы оценки энтропии печально известны своей ненадежностью, противник может их легко обмануть (именно поэтому в системе Fortuna авторы отказались от оценивания энтропии, присутствовавшего в Yarrow). Во-вторых, у */dev/random* довольно быстро заканчивается оцененная энтропия, что может привести к отказу от обслуживания, поскольку приложения должны будут ждать накопления новой энтропии. Таким образом, на практике */dev/random* ничем не лучше */dev/urandom*, но создает больше проблем, чем решает.

Оценивание энтропии в /dev/random

В Linux наблюдать, как изменяется оценка энтропии в */dev/random*, можно, читая ее текущее значение, выраженное в битах, из файла */proc/sys/kernel/random/entropy_avail*. Так, скрипт оболочки в листинге 2.3 сначала доводит оценку энтропии до минимума, прочитав 4 КБ из */dev/random*, потом ждет, пока оценка не достигнет 128 бит, читает из */dev/random* 64 бита и показывает новую оценку. Запустив этот скрипт, обратите внимание, что действия пользователя ускоряют восполнение энтропии (прочитанные байты печатаются на stdout в кодировке base64).

Листинг 2.3. Скрипт, показывающий, как изменяется оценка энтропии в /dev/random

```
#!/bin/sh
ESTIMATE=/proc/sys/kernel/random/entropy_avail
timeout 3s dd if=/dev/random bs=4k count=1 2> /dev/null | base64
ent=`cat $ESTIMATE`
while [ $ent -lt 128 ]
do
    sleep 3
    ent=`cat $ESTIMATE`
    echo $ent
done
dd if=/dev/random bs=8 count=1 2> /dev/null | base64
cat $ESTIMATE
```

В примере прогона в листинге 2.3 показан вывод скрипта в листинге 2.4. (Угадайте, когда я начал хаотично двигать мышью и стучать по клавиатуре для сбора энтропии.)

Листинг 2.4. Пример выполнения скрипта, демонстрирующего изменение оценки энтропии

```
xFNX/f2R87/zrrNJ6Ibr5R1L913tl+F4GNzKb60BC+qQnHQcyA==
2
18
19
27
28
72
124
193
jq8XWCt8
129
```

Как видно из листинга 2.4, согласно оценке */dev/random*, в пуле осталось 193 – 64 = 129 бит энтропии. Имеет ли смысл считать, что в PRNG стало на *N* бит энтропии меньше, лишь потому что из PRNG только что было прочитано *N* бит? (Подсказка: нет.)

Примечание *Как и* `/dev/random`*, системный вызов Linux* `getrandom()` *блокирует выполнение, если не набрано достаточно начальной энтропии. Но, в отличие от* `/dev/random`*, он не пытается оценить количество энтропии в системе и никогда не блокируется после стадии инициализации. И это хорошо. (Путем установки флагов можно заставить* `getrandom()` *использовать* `/dev/random` *и блокироваться, но я не вижу, зачем бы это могло понадобиться.)*

Функция CryptGenRandom() в Windows

В Windows унаследованный интерфейс к системному PRNG, работающий в пространстве пользователя, дает функция `CryptGenRandom()` из криптографического (API). В современных версиях Windows функция `CryptGenRandom()` заменена функцией `BcryptGenRandom()`, входящей в состав Cryptography API: Next Generation (CNG). В Windows PRNG получает энтропию от драйвера ядра *cng.sys* (раньше *ksecdd.sys*), основанного на идеях Fortuna. Как обычно в Windows, процесс получения случайных битов сложен.

В листинге 2.5 показан типичный вызов `CryptGenRandom()` из программы на C++ со всеми необходимыми проверками.

Листинг 2.5. Использование интерфейса `CryptGenRandom()` *к PRNG в Windows*

```
int random_bytes(unsigned char *out, size_t outlen)
{
    static HCRYPTPROV handle = 0; /* освобождается только по завершении */
                                  /* программы */
    if(!handle) {
        if(!CryptAcquireContext(&handle, 0, 0, PROV_RSA_FULL,
                                CRYPT_VERIFYCONTEXT | CRYPT_SILENT)) {
            return -1;
        }
    }
    while(outlen > 0) {
        const DWORD len = outlen > 1048576UL ? 1048576UL : outlen;
        if(!CryptGenRandom(handle, len, out)) {
            return -2;
        }
        out += len;
        outlen -= len;
    }
    return 0;
}
```

Обратите внимание, что в листинге 2.5 перед обращением к самому PRNG необходимо объявить *поставщик служб шифрования* (`HCRYPTPROV`), а затем получить криптографический контекст с помощью функции `CryptAcquireContext()`. Поэтому количество мест, где возможна ошибка, увеличивается. Например, в окончательной версии программы шифрования `TrueCrypt` не проверялось, завершается ли ошибкой вызов `CryptAcquireContext()`, и в результате случайность без ведома пользователя могла оказаться неоптимальной. По счастью, новый интерфейс `BCryptGenRandom()` гораздо проще, в нем не требуется явно открывать описатель (по крайней мере, его легче использовать без описателя).

Аппаратный PRNG: RDRAND в микропроцессорах Intel

Пока что мы обсуждали только программные PRNG, а теперь рассмотрим аппаратный. *Цифровой генератор случайных чисел Intel* представляет собой аппаратный PRNG, впервые включенный в микроархитектуру Intel Ivy Bridge в 2012 году. Он основан на рекомендациях NIST SP 800-90 и использует шифр Advanced Encryption Standard (AES) в режиме CTR_DRBG. Доступ к PRNG от Intel осуществляется с помощью ассемблерной команды `RDRAND`, которая предлагает независимый от операционной системы интерфейс, принципиально более быстрый, чем программные PRNG.

Если программные PRNG пытаются получить энтропию из непредсказуемых источников, то у `RDRAND` имеется единственный источник энтропии, который поставляет поток данных, состоящий из нулей и единиц. В электронике этот источник называется двойной дифференциальной защелкой с обратной связью; по существу, это небольшая схема, которая переключается между двумя состояниями (0 или 1) в зависимости от флуктуаций теплового шума с частотой 800 МГц. Такой источник обычно достаточно надежен.

Команда `RDRAND` принимает в качестве аргумента 16-, 32- или 64-разрядный регистр и записывает в него случайное значение. Она устанавливает флаг переноса в 1, если в указанный регистр действительно помещено случайное значение, и 0 в противном случае, поэтому если вы пишете ассемблерный код напрямую, то необходимо проверять флаг `CF`. Отметим, что внутренние функции современных компиляторов C не проверяют флаг `CF`, а возвращают его значение.

Примечание *В инфраструктуре Intel PRNG, помимо* `RDRAND`*, имеется еще ассемблерная команда* `RDSEED`*, которая возвращает случайные биты непосредственно от источника энтропии, после некоторой криптографической обработки. Она предназначена для поставки начальных значений другим PRNG.*

Intel PRNG документирован не полностью, но основан на известных стандартах и был подвергнут аудиту со стороны авторитетной компании Cryptography Research (см. ее отчет «Analysis of Intel's Ivy Bridge Digital Random Number Generator»). Тем не менее существуют сомнения в его безопасности, особенно после разоблачений криптографических закладок Сноуденом. Действительно, PRNG – идеальная мишень для подрывных действий. Если вас это беспокоит, но вы все-таки хотите использовать команды `RDRAND` или `RDSEED`, то просто добавьте к ним другие источники энтропии. Это предотвратит эффективную эксплуатацию гипотетической закладки в оборудовании Intel или связанном с ними микрокоде во всех сценариях, кроме самых продвинутых.

Какие возможны проблемы

В заключение я покажу несколько примеров ошибок из-за неправильной работы с генераторами случайных чисел. Выбирать тут есть из чего, но я остановился на примерах, которые просты для понимания и иллюстрируют различные проблемы.

Плохие источники энтропии

В 1996 году в реализации SSL в браузере Netscape вычисление 128-битовых начальных значений PRNG было основано на псевдокоде, показанном в листинге 2.6 (скопирован из статьи Голдберга и Вагнера по адресу *http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html*).

Листинг 2.6. Псевдокод генерирования 128-битовых начальных значений PRNG в браузере Netscape

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* время, прошедшее с 1970 года */
    pid = process ID; ppid = parent process ID;
    a = mklcpr(microseconds);
  ❶ b = mklcpr(pid + seconds + (ppid << 12));
    seed = MD5(a, b); /* получение 128-битового значения */
                      /* с помощью функции хеширования MD5 */

mklcpr(x) /* с точки зрения криптографии, не важна; показана для полноты */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* очень хорошая стандартная функция перемешивания, исходный код опущен */
```

Проблема здесь в том, что идентификаторы процессов и число микросекунд – значения, которые можно угадать. В предположении, что значение `seconds` известно, существует всего 10^6 возможных значений `microseconds`, поэтому энтропия равна $\log(10^6)$, т. е. приблизительно 20 бит. Идентификатор процесса (PID) и родительского процесса (PPID) – 15-битовые значения, что дает еще 15 + 15 = 30 бит энтропии. Но, внимательно приглядевшись к способу вычисления `b` в точке ❶, мы увидим, что перекрытие трех бит оставляет всего лишь около 15 + 12 = 27 бит энтропии, так что полная энтропия равна только 47 бит, тогда как в 128-битовом начальном значении должно быть 128 бит энтропии.

Недостаточная энтропия на этапе начальной загрузки

В 2012 году исследователи просканировали весь интернет и собрали коллекцию открытых ключей из TLS-сертификатов SSH-хостов. Обнаружилось несколько систем с одинаковыми открытыми ключа-

ми, а в некоторых случаях ключи были очень похожи (точнее, ключи RSA с общими простыми множителями): короче говоря, пары чисел $n = pq$ и $n' = p'q'$, где $p = p'$, тогда как в идеале все p и q должны быть различны.

Изучение проблемы показало, что многие устройства генерируют свои открытые ключи при первой начальной загрузке, когда еще не набралось достаточно энтропии, хотя во всех остальных отношениях используемые PRNG (как правило, */dev/urandom*) не вызывали нареканий. В итоге PRNG в различных системах порождали одинаковые случайные биты, поскольку базовый источник энтропии был один и тот же (например, зашитое в код начальное значение).

Не вдаваясь в детали, можно сказать, что идентичные ключи появляются, потому что схема генерации ключей устроена, как показано в следующем псевдокоде:

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
n = p*q
```

Если две системы выполнят этот код с одинаковым значением `seed`, то они вычислят одинаковые значения p, q и n.

Общие простые множители в разных ключах появляются, когда в схему генерации ключей включается дополнительная энтропия, как в псевдокоде ниже:

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_entropy()
q = prng.generate_random_prime()
n = p*q
```

Если две системы выполнят этот код с одинаковым значением `seed`, то они вычислят одинаковые значения p, но благодаря включению энтропии в строке `prng.add_entropy()` значения q будут различны.

Проблема в том, что, зная $n = pq$ и $n' = pq'$, можно без труда восстановить p, вычислив *наибольший общий делитель* (НОД, *англ.* GCD) n и n'. Детали см. в статье Heninger, Durumeric, Wustrow, and Halderman «Mining Your Ps and Qs», доступной на сайте *https://factorable.net/*.

Криптографически нестойкие PRNG

Выше мы обсудили различие между криптографически стойкими и нестойкими PRNG и объяснили, почему последние никогда не следует использовать в криптографических приложениях. Увы, многие системы пренебрегают этим предостережением, поэтому я чувствую себя обязанным привести хотя бы один такой пример.

Популярное приложение MediaWiki работает в Википедии и на многих других вики-сайтах. Случайность используется в нем для генерирования маркеров безопасности и временных паролей, которые, конечно же, должны быть непредсказуемы. К сожалению, в теперь уже устаревшей версии MediaWiki для этой цели использовался криптографически нестойкий PRNG, вихрь Мерсенна. Ниже приведен фрагмент уязвимого исходного кода MediaWiki. Взгляните на функцию, вызываемую для получения случайного бита, и прочитайте комментарии.

```
      /**
      * Генерирует кажущийся случайным маркер hex-у для различных целей.
      * Могла бы быть сделана более криптографически стойкой, если кому-нибудь
      * это надо.
      * @return string
      */
function generateToken( $salt = '' ) {
    $token = dechex(mt_rand()).dechex(mt_rand());
    return md5( $token . $salt );
}
```

Вы обратили внимание на вызов `mt_rand()`? Здесь `mt` означает Mersenne Twister (вихрь Мерсенна), криптографически нестойкий PRNG, обсуждавшийся ранее. В 2012 году исследователи показали, как можно эксплуатировать предсказуемость вихря Мерсенна, чтобы предсказать будущие маркеры и временные пароли, если известно два маркера безопасности. Приложение MediaWiki было исправлено, теперь в нем используется криптографически стойкий PRNG.

Дефектная выборка при стойком PRNG

Следующая ошибка показывает, что даже криптографически стойкий PRNG с достаточной энтропией может порождать смещенное распределение. Программа организации чатов Cryptocat проектировалась для безопасного общения. В ней использовалась функция, которая должна была создавать строку равномерно распределенных десятичных цифр, т. е. цифр от 0 до 9. Однако брать случайные байты по модулю 10 недостаточно для получения равномерного распределения, потому что при делении чисел от 0 до 255 на 10 количество различных остатков (чисел от 0 до 9) неодинаково.

Для решения этой проблемы авторы Cryptocat поступили следующим образом:

```
Cryptocat.random = function() {
    var x, o = '';
    while (o.length < 16) {
        x = state.getBytes(1);
        if (x[0] <= 250) {
```

```
            o += x[0] % 10;
        }
    }
    return parseFloat('0.' + o)
}
```

Это почти правильно. Поскольку количество рассматриваемых чисел кратно 10 (а остальные просто отбрасываются), то можно ожидать равномерного распределения цифр от 0 до 9. К сожалению, в условие `if` вкралась ошибка на единицу. Оставляю анализ деталей вам в качестве упражнения. Вы должны обнаружить, что энтропия сгенерированных значений равна приблизительно 45, а не 53 бита (указание: вместо <= следует использовать <).

Для дополнительного чтения

В этой главе я смог рассмотреть лишь самые начатки темы случайности в криптографии. О теории случайности написано очень много, в т. ч. о различных понятиях энтропии, экстракторах случайности и даже о механизмах рандомизации и дерандомизации в теории сложности. Если вы хотите больше узнать о PRNG и их безопасности, прочитайте классическую статью Kelsey, Schneier, Wagner, and Hall «Cryptanalytic Attacks on Pseudorandom Number Generators», опубликованную в 1998 году. Затем изучите реализации PRNG в своих любимых приложениях и попытайтесь найти в них слабые места (поиск по запросу «random generator bug» даст кучу примеров).

Но мы не расстаемся со случайностью. Она будет снова и снова возникать на страницах этой книги, и вы увидите многочисленные примеры ее применения для построения безопасных систем.

3

КРИПТОГРАФИЧЕСКАЯ БЕЗОПАСНОСТЬ

В криптографии безопасность определяется не так, как в общей информатике. Основное различие между безопасностью ПО и криптографической безопасностью заключается в том, что последняя поддается *количественному измерению*. В отличие от мира ПО, где приложение обычно считается либо безопасным, либо небезопасным, в мире криптографии часто можно оценить, сколько усилий придется приложить для взлома криптографического алгоритма. Кроме того, если основная цель безопасности ПО – помешать противнику использовать код программы для нанесения ущерба, то цель криптографической безопасности – сделать невозможным решение четко поставленной задачи.

В криптографических задачах встречаются математические понятия, но нет сложной математики – по крайней мере, не в этой книге. В этой главе мы познакомимся с некоторыми понятиями безопасности и их применением к решению реальных задач. В следующих

разделах мы обсудим теоретически безупречные и практически полезные способы количественного измерения криптографической безопасности. Мы поговорим об информационной и вычислительной безопасности, о безопасности на уровне битов и полной стоимости атаки, о доказуемой и эвристической безопасности, а также о генерировании симметричных и асимметричных ключей. И закончим главу примерами ошибок в казалось бы строгой криптографии.

Определение невозможного

В главе 1 я описал безопасность шифра относительно целей и возможностей противника и назвал шифр безопасным, если достижение противником его целей невозможно при известных располагаемых средствах. Но что в этом контексте означает слово «невозможно»?

В криптографии есть два определения понятия невозможного: информационная безопасность и вычислительная безопасность. Грубо говоря, *информационная безопасность* – это теоретическая невозможность, а *вычислительная безопасность* – практическая невозможность. Для информационной безопасности не нужно количественно измерять безопасность, потому что шифр рассматривается как безопасный или небезопасный – середины быть не может. Поэтому на практике это понятие бесполезно, хотя играет важную роль в теоретической криптографии. Вычислительная безопасность – более практически полезная мера стойкости шифра.

Безопасность в теории: информационная безопасность

Информационная безопасность характеризует не трудность взлома шифра, а можно ли его взломать вообще. Шифр считается информационно безопасным, только если его нельзя взломать даже при наличии неограниченного времени и памяти. Если успешная атака на шифр возможна, хотя и потребует триллион лет, то такой шифр информационно *не* безопасен.

Например, одноразовый блокнот, описанный в главе 1, информационно безопасен. Напомним, что одноразовый блокнот преобразует открытый текст P в шифртекст $C = P \oplus K$, где K – случайная битовая строка, уникальная для каждого открытого текста. Этот шифр информационно безопасен, потому что, имея шифртекст и неограниченное время для перебора всех возможных ключей K и вычисления соответствующего открытого текста P, мы все равно не смогли бы найти правильный K, потому что возможных P столько же, сколько K.

Безопасность на практике: вычислительная безопасность

В отличие от информационной безопасности, при оценке вычислительной безопасности шифр считается безопасным, если его нельзя

взломать за *разумное* время и при наличии разумных ресурсов: памяти, оборудования, бюджета, энергии и т. д. Вычислительная безопасность – это способ количественного измерения безопасности шифра или вообще любого криптографического алгоритма.

Например, рассмотрим шифр **E**, для которого известна пара «открытый текст – шифртекст» (*P*, *C*), но неизвестен 128-битовый ключ *K*, с помощью которого вычислен $C = \mathbf{E}(K, P)$. Этот шифр не является информационно безопасным, потому что мы могли бы взломать его, перебрав все 2^{128} возможных 128-битовых ключей *K*, пока не найдем такой, для которого $\mathbf{E}(K, P) = C$. Но на практике, даже если мы будем проверять по 100 млрд ключей в секунду, для решения задачи потребуется более 100 000 000 000 000 000 000 лет. Иными словами, разумно считать, что этот шифр вычислительно безопасен, потому что взломать его практически невозможно.

Вычислительную безопасность иногда характеризуют двумя величинами:

- *t* – предельное число операций, которое может выполнить противник;
- ε («эпсилон») – предельная вероятность успешной атаки.

Говорят, что криптографическая схема является (*t*, ε)-*безопасной*, если, выполнив не более *t* операций (не важно, каких), противник не сможет добиться вероятности успеха более ε, где ε – число от 0 до 1. Вычислительная безопасность определяет предел трудности взлома криптографического алгоритма.

Тут важно отметить, что *t* и ε – всего лишь предельные значения: если шифр является (*t*, ε)-*безопасным*, то никакой противник, выполнивший менее *t* операций, не сможет добиться успеха (с вероятностью ε). Но это не значит, что противник, выполнивший ровно *t* операций, добьется успеха, и ничего не говорит о том, сколько же операций необходимо, а их число может быть гораздо больше *t*. Мы говорим, что *t* – *нижняя граница* потребных вычислительных усилий, поскольку для компрометации системы необходимо по меньшей мере *t* операций.

Иногда точно известно, сколько усилий нужно приложить для взлома шифра; в таких случаях говорят, что (*t*, ε)-*безопасность* дает *точную границу*, если существует атака, которая приводит к взлому шифра с вероятностью ε ровно за *t* операций.

Например, рассмотрим симметричный шифр со 128-битовым ключом. В идеале этот шифр должен быть $(t, t/2^{128})$-*безопасным* для любого значения *t* между 1 и 2^{128}. Наилучшей атакой должен быть *полный перебор* (т. е. испытание всех ключей, пока не будет найден правильный). Любая более эффективная атака должна была бы использовать какое-то упущение в шифре, поэтому мы стремимся создавать шифры, для которых полный перебор является наилучшей из возможных атак.

В предположении $(t, t/2^{128})$-*безопасности* рассмотрим вероятность успеха трех возможных атак.

- В первом случае $t = 1$, т. е. противник проверяет один ключ и достигает успеха с вероятностью $\varepsilon = 1/2^{128}$.
- Во втором случае $t = 2^{128}$, т. е. противник проверяет все 2^{128} ключей, и один из них точно правилен. Таким образом, вероятность $\varepsilon = 1$ (если противник перепробует все ключи, то, очевидно, найдет правильный).
- В третьем случае противник пробует только $t = 2^{64}$ ключей и добивается успеха с вероятностью $\varepsilon = 2^{64}/2^{128} = 2^{-64}$. Если перебирается лишь часть всего множества ключей, то вероятность успеха пропорциональна количеству проверенных ключей.

Мы можем заключить, что шифр с n-битовым ключом в лучшем случае является $(t, t/2^n)$-*безопасным* для любого t от 1 до 2^n, потому что вне зависимости от стойкости шифра атака полным перебором всегда завершится успешно. Поэтому ключ должен быть достаточно длинным, чтобы сделать атаки полным перебором практически неосуществимыми.

Примечание *В этом примере мы подсчитываем количество вычислений шифра, а не абсолютное время или число процессорных тактов. Вычислительная безопасность не зависит от технологии, и это правильно: шифр, являющийся* (t, ε)*-безопасным сегодня, останется таким и завтра, однако то, что считается сегодня безопасным, завтра может перестать быть таковым.*

Количественное измерение безопасности

Обнаружив атаку, мы первым делом хотим узнать, насколько она эффективна теоретически и осуществима ли она на практике. Аналогично для предположительно безопасного шифра мы хотим знать, насколько настойчивым усилиям он может противостоять. Чтобы ответить на эти вопросы, я объясню, как можно измерить криптографическую безопасность в битах (теоретический взгляд на вещи) и какие факторы влияют на фактическую стоимость атаки.

Измерение безопасности в битах

В контексте вычислительной безопасности шифр называется *t-безопасным*, если для успешной атаки необходимо по меньшей мере t операций. Таким образом, мы избегаем интуитивно неочевидной нотации (t, ε), предполагая, что вероятность успеха ε близка к 1 – ведь именно этот случай интересует нас на практике. Далее безопасность выражается в битах; выражение «n-битовая безопасность» означает, что для компрометации некоторого аспекта безопасности необходимо приблизительно 2^n операций.

Если примерно известно, сколько операций требуется для взлома шифра, то его уровень безопасности в битах можно определить,

вычислив двоичный логарифм числа операций: если требуется 1 000 000 операций, то уровень безопасности равен $\log_2(1\,000\,000)$, или около 20 бит (1 000 000 приближенно равно 2^{20}). Напомним, что n-битовый ключ обеспечивает не более чем n-битовую безопасность, потому что атака с полным перебором всех 2^n возможных ключей всегда будет успешной. Но размер ключа не всегда равен уровню безопасности – это лишь *верхняя граница*, или максимально возможный уровень безопасности.

Уровень безопасности может быть меньше размера ключа по двум причинам:

- для взлома шифра достаточно меньшего числа операций, чем ожидалось, – например, если для нахождения ключа нужно просмотреть не все 2^n вариантов, а только их часть;
- уровень безопасности шифра намеренно сделан меньшим, чем размер ключа, как в большинстве алгоритмов с открытым ключом. Например, алгоритм RSA с 2048-битовым закрытым ключом обеспечивает безопасность меньше 100 бит.

Битовая безопасность полезна при сравнении уровней безопасности шифров, но не дает достаточно информации о фактической стоимости атаки. Иногда эта абстракция оказывается слишком простой, потому что предполагается, что для взлома шифра с уровнем безопасности n бит хватает 2^n операций, но что это за операции, не конкретизируется. Поэтому реальные уровни безопасности двух шифров с одинаковой битовой безопасностью могут сильно различаться по реальной стоимости атаки для противника.

Пусть имеется два шифра со 128-битовой безопасностью и 128-битовым ключом. Чтобы взломать тот и другой, необходимо вычислить шифр 2^{128} раз, но вычисление второго шифра занимает в 100 раз больше времени, чем первого. Тогда на 2^{128} вычислений второго шифра уйдет примерно столько же времени, сколько на $100 \times 2^{128} \approx 2^{134.64}$ вычислений первого. Если подсчитывать в терминах более быстрого первого шифра, то для взлома более медленного требуется $2^{134.64}$ операций. А если подсчитывать в терминах второго шифра, то только 2^{128} операций. Правильно ли будет сказать, что второй шифр более стойкий, чем первый? В принципе, да, но стократное различие в производительности широко используемых шифров встречается редко.

Неточность определения операции создает много трудностей при сравнении эффективности атак. В описании некоторых атак утверждается, что им удалось снизить безопасность шифра, потому что выполняется 2^{120} вычислений некоторой операции, а не 2^{128} вычислений шифра, но при этом упускается из виду скорость каждого типа атак. Не всегда атака с 2^{120} операциями работает быстрее, чем атаки с полным перебором 2^{128} вариантов.

Тем не менее битовая безопасность остается полезным понятием, если только операция определена разумным образом, т. е. занимает примерно столько же времени, сколько вычисление шифра. В конце

концов, в реальной жизни уровень безопасности достаточно определить с точностью до порядка величины.

Полная стоимость атаки

Битовая безопасность отражает стоимость самой быстрой атаки на шифр путем оценивания количества необходимых для успеха операций по порядку величины. Но на стоимость атаки влияют и другие факторы, которые нужно принимать во внимание, оценивая фактический уровень безопасности. Я объясню четыре главных фактора: параллелизм, память, предварительные вычисления и количество мишеней.

Параллелизм

Прежде всего нужно учитывать вычислительный параллелизм. Рассмотрим, к примеру, две атаки, насчитывающие 2^{56} операций каждая. Разница между ними в том, что вторую атаку можно распараллелить, а первую нет: в первой атаке нужно выполнить 2^{56} *последовательно зависимых операций* типа $x_{i+1} = f_i(x_i)$ для некоторого x_0 и некоторых функций f_i (где i изменяется от 1 до 2^{56}), а во второй 2^{56} *независимых* операций типа $x_i = f_i(x)$ для некоторого x и i от 1 до 2^{56}, которые могут производиться параллельно. Параллельная обработка может быть на несколько порядков быстрее последовательной. Например, если бы нам было доступно $2^{16} = 65\,536$ процессоров, то рабочую нагрузку параллельной атаки можно было бы разбить на 2^{16} независимых задач, каждая из которых выполняет $2^{56}/2^{16} = 2^{40}$ операций. Но первая атака ничего не выигрывает от наличия нескольких ядер, потому что каждая операция зависит от результата предыдущей. Поэтому параллельная атака завершится в 65 536 быстрее последовательной, хотя количество выполненных операций одно и то же.

Примечание *Алгоритмы, которые работают в N раз быстрее, когда атака реализуется на N ядрах, называются естественно параллельными, и мы говорим, что их время выполнения линейно масштабируется относительно количества вычислительных ядер.*

Память

Второй фактор, который следует учитывать при определении стоимости атаки, – располагаемая память. При оценивании криптоаналитических атак нужно принимать во внимание время и память: сколько операций они производят в единицу времени, сколько потребляют памяти, как используется эта память и каково быстродействие доступной памяти? К сожалению, битовая безопасность отражает только время, потребное для выполнения атаки.

Говоря о способе использования памяти, важно учитывать количество обращений к памяти, необходимых для совершения атаки, скорость обращения к памяти (помня, что скорости записи и чтения

могут различаться), размер читаемых или записываемых данных, характер доступа (последовательный или произвольный) и организацию данных в памяти. Например, на одном из современных CPU общего назначения чтение из регистра занимает один такт, чтение из процессорного кеша (кеша L3) – примерно 20 тактов, а чтение из DRAM – по меньшей мере 100 тактов. Коэффициент 100 – это разница между одним днем и тремя месяцами.

Предварительные вычисления

Предварительные вычисления выполняются один раз, но их результатами можно пользоваться многократно при последующих атаках. Иногда это называется *офлайновой стадией* атаки.

Например, рассмотрим атаку с компромиссом между временем и памятью. В этом случае противник предварительно выполняет очень большое вычисление гигантских справочных таблиц, которые сохраняются и используются в ходе фактической атаки. Так, в одной атаке на шифрование в мобильной сети 2G пришлось потратить два месяца на построение таблиц общим объемом 2 терабайта, которые затем использовались для взлома шифра и нахождения секретного сеансового ключа за несколько секунд.

Количество мишеней

И наконец, мы подошли к количеству мишеней атаки. Чем больше количество мишеней, тем шире поверхность атаки и тем больше информации противник может получить о ключах, за которыми охотится.

Например, рассмотрим поиск ключа полным перебором: если мы ищем всего один n-битовый ключ, то для гарантированного его нахождения потребуется 2^n попыток. Если мы ищем несколько n-битовых ключей, скажем M, и если для одного P имеется M различных шифртекстов, где $C = \mathbf{E}(K, P)$ для каждого из M искомых ключей (K), то для нахождения всех ключей опять-таки понадобится 2^n попыток. Но если нас интересует *хотя бы один* из M ключей, а не все сразу, то для успеха в среднем достаточно $2^n / M$ попыток. Например, чтобы взломать один 128-битовый ключ из $2^{16} = 65\,536$ ключей-мишеней, в среднем нужно произвести $2^{128-16} = 2^{112}$ вычислений шифра. Следовательно, стоимость (и время проведения) атаки уменьшается при возрастании числа мишеней.

Выбор и вычисление уровней безопасности

Выбор уровня безопасности часто сводится к выбору между 128-битовой и 256-битовой безопасностью, потому что большинство стандартных криптографических алгоритмов и их реализаций доступны для одного из этих двух уровней. Есть также схемы с 64- или 80-битовой безопасностью, но в современном мире они уже не считаются достаточно безопасными.

На верхнем уровне 128-битовая безопасность означает, что для взлома криптосистемы нужно выполнить приблизительно 2^{128} операций. Чтобы вы поняли, насколько велико это число, скажу, что возраст Вселенной равен примерно 2^{88} наносекунд (в одной секунде миллиард наносекунд). Поскольку при современных технологиях для проверки одного ключа нужно не менее одной наносекунды, атака займет в несколько раз больше времени, чем существует Вселенная (в 2^{40} раз больше, если быть точным).

Но, быть может, параллелизм и выбор нескольких мишеней могут существенно уменьшить время атаки? Вряд ли. Допустим, нас интересует взлом любой из миллиона мишеней, и в нашем распоряжении имеется миллион параллельных ядер. Тогда время поиска уменьшится с 2^{128} до $(2^{128}/2^{20})/2^{20} = 2^{88}$, что эквивалентно одному времени жизни Вселенной.

При оценке уровня безопасности следует также помнить о развитии технологий. Закон Мура утверждает, что эффективность вычислений удваивается примерно каждые два года. Можно интерпретировать этот факт как потерю одного бита безопасности в два года: если сегодня бюджет в 1000 долларов позволяет взломать 40-битовый ключ за час, то, по закону Мура, через два года при том же бюджете можно будет взломать 41-битовый ключ (я намеренно упрощаю). Экстраполируя, можно сказать, что через 80 лет безопасность станет на 40 бит меньше. Иными словами, через 80 лет выполнение 2^{128} операций может стоить столько же, сколько сегодня стоит выполнение 2^{88} операций. С учетом параллелизма и нескольких мишеней время вычислений уменьшается до 2^{48} наносекунд, или примерно трех дней. Но такая экстраполяция в высшей степени неточна, потому что закон Мура не может и не будет масштабироваться настолько прямолинейно. Тем не менее идея понятна: то, что кажется неосуществимым сегодня, вполне может стать реальным через сто лет.

Иногда уровень безопасности ниже 128 бит вполне оправдан. Например, если данные должны оставаться в безопасности лишь короткое время, а затраты на реализацию более высокого уровня негативно отразятся на стоимости или удобстве работы с системой. Реальный пример – платное телевидение, в котором ключи шифрования имеют длину 48 или 64 бита. На первый взгляд, до смешного мало, но такой уровень безопасности достаточен, потому что ключ обновляется каждые 5 или 10 секунд.

Тем не менее для обеспечения долговременной безопасности следует выбирать 256-битовый или чуть меньший уровень. Даже в худшем случае – при появлении квантовых компьютеров (см. главу 14) – схема с 256-битовой безопасностью вряд ли будет взломана в обозримом будущем. Более 256 бит практически не нужно, это всего лишь маркетинговый ход.

Работающий в NIST криптограф Джон Келси когда-то сказал: «Разница между поиском 80-битового и 128-битового ключей такая же, как разница между полетом на Марс и на альфу Центавра. Но, насколько я понимаю, с точки зрения практических атак полным пере-

бором, между 192-битовым и 256-битовым ключами нет существенного различия: невозможно – значит, невозможно».

Достижение безопасности

После того как уровень безопасности выбран, важно гарантировать, что криптографическая схема отвечает ему. Иначе говоря, необходима *уверенность*, а не просто надежда, что все будет работать, как запланировано, – всегда.

Для обретения уверенности в безопасности криптографического алгоритма можно опираться на математические доказательства – это называется *доказуемой* (provable) *безопасностью*, или на свидетельства, основанные на безуспешных попытках взломать алгоритм, – я называю это *эвристической безопасностью*, хотя иногда можно встретить термин *предположительная* (probable) безопасность. Эти два подхода дополняют друг друга, и, как мы увидим ниже, ни один не является более предпочтительным.

Доказуемая безопасность

Говоря о доказуемой безопасности, мы имеем в виду, что существует возможность строго доказать, что взломать криптографическую схему так же трудно, как решить другую заведомо трудную задачу. Такое *доказательство безопасности* гарантирует, что схема останется безопасной до тех пор, пока трудная задача остается трудной. Доказательство такого типа называется *сведением*, оно уходит корнями в теорию сложности. Мы говорим, что взлом некоторого шифра сводится к задаче X, если любой метод решения X дает также метод взлома шифра.

Доказательства безопасности бывают двух видов в зависимости от характера предположительно трудной задачи: относительно математической задачи и относительно криптографической задачи.

Доказательства относительно математической задачи

Многие доказательства безопасности (например, для криптографии с открытым ключом) показывают, что взлом криптографической схемы по меньшей мере так же труден, как решение некоторой трудной математической задачи. Речь идет о задачах, для которых решение заведомо существует и легко проверить, что предъявленное решение действительно является таковым, но найти решение трудно с вычислительной точки зрения.

Примечание *Не существует настоящих доказательств того, что кажущиеся трудными математические задачи действительно трудны. На самом деле доказательство этого факта для некоторого класса задач – одна из самых важных проблем в теории сложности, и на момент написания книги Математический институт Клэя объявил награду в миллион долларов за ее решение. Подробнее мы будем обсуждать эту тему в главе 9.*

Например, рассмотрим решение *задачи факторизации*, одной из самых известных математических задач в криптографии: дано число, про которое известно, что оно является произведением двух простых чисел ($n = pq$), и требуется найти эти числа. Например, если $n = 15$, то множителями будут 3 и 5. Для небольших чисел это легко, но с ростом числа трудность задачи экспоненциально возрастает. Например, для числа n длиной 3000 бит (около 900 десятичных цифр) или более задача факторизации считается практически неразрешимой.

Одной из самых известных схем, опирающихся на задачу факторизации, является RSA. В этом случае открытый текст P, рассматриваемый как большое число, шифруется путем вычисления $C = P^e \bmod n$, где число e и $n = pq$ составляют открытый ключ. При дешифрировании открытый текст восстанавливается по шифртексту путем вычисления $P = C^d \bmod n$, где d – закрытый ключ, ассоциированный с e и n. Если мы сможем факторизовать n, то взломаем схему RSA (поскольку сможем найти закрытый ключ, зная открытый). И наоборот, зная закрытый ключ, мы сможем факторизовать n. Таким образом, нахождение закрытого ключа RSA и факторизация n – эквивалентные трудные задачи. Именно такого рода сведение интересует нас при доказательстве безопасности. Однако нет никакой гарантии, что реконструкция открытого текста в RSA – задача столь же трудная, как факторизация n, поскольку знания открытого текста недостаточно для нахождения закрытого ключа.

Доказательства относительно другой криптографической задачи

Вместо того чтобы сравнивать криптографическую схему с математической задачей, можно сравнить ее с другой криптографической схемой и доказать, что вторую можно взломать, только если возможно взломать первую. Доказательства безопасности симметричных шифров обычно строятся именно таким образом.

Например, имея всего один перестановочный алгоритм, можно построить симметричные шифры, генераторы случайных битов и другие криптографические объекты, например функции хеширования. Для этого нужно только сочетать перестановки с различными типами входных данных (как мы увидим в главе 6). Тогда доказательства показывают, что все созданные схемы безопасны, если безопасен перестановочный алгоритм. Иными словами, мы точно знаем, что новый алгоритм *не слабее* исходного. Такие доказательства обычно строятся путем конструирования атаки на меньший компонент, если известна атака на больший, т. е. посредством сведения.

При доказательстве того, что один криптоалгоритм не слабее другого, главным преимуществом является уменьшенная поверхность атаки: вместо того чтобы анализировать базовый алгоритм и комбинацию, мы можем просто рассмотреть базовый алгоритм нового шифра. Точнее, разрабатывая шифр, в котором используется новый перестановочный алгоритм и новая комбинация, можно доказать,

что комбинация не уменьшает безопасность по сравнению с базовым алгоритмом. Поэтому, чтобы взломать комбинацию, нужно взломать новый перестановочный алгоритм.

Подводные камни

Криптографы часто прибегают к доказательствам безопасности – как относительно математических схем, так и относительно других криптографических схем. Но существование доказательства безопасности еще не гарантирует идеальность криптографической схемы и не может служить основанием для пренебрежения более практическими аспектами реализации. Как писал криптограф Ларс Кнудсен, «то, что доказуемо безопасно, вероятно, таковым не является», имея в виду, что доказательство безопасности не следует воспринимать как абсолютную гарантию безопасности. Хуже того, существует несколько причин, из-за которых «доказуемо безопасная» схема может приводить к утрате безопасности.

Одна из таких причин кроется в самой фразе «доказательство безопасности». В математике доказательством считается демонстрация *абсолютной истины*, тогда как в криптографии доказательство демонстрирует лишь *относительную истину*. Например, доказательство того, что взломать шифр так же трудно, как вычислить дискретный логарифм – найти число x, если известны g и $g^x \bmod n$, – гарантирует лишь, что если ваш шифр взламывается, то взламывается и много других шифров, и никто не станет вас винить, если произойдет худшее.

Еще один подводный камень заключается в том, что безопасность обычно доказывается в отношении какого-то одного аспекта. Например, можно было бы доказать, что найти закрытый ключ шифра так же трудно, как решить задачу факторизации. Но если можно восстановить открытый текст по шифртексту, не зная ключа, то это доказательство гроша ломаного не стоит, поскольку находить ключ вообще не нужно.

Кроме того, доказательства не всегда корректны, и может оказаться, что взломать алгоритм проще, чем думали раньше.

Примечание *К сожалению, немногие ученые внимательно проверяют доказательства, которые часто занимают десятки страниц, и это затрудняет контроль качества. Впрочем, демонстрация некорректности доказательства еще не означает, что доказываемое утверждение совершенно неверно; если результат правилен, то доказательство иногда можно спасти, исправив ошибки.*

Следует также иметь в виду, что иногда трудную математическую задачу оказывается легче решить, чем предполагалось. Например, при некоторых слабых параметрах взломать алгоритм RSA нетрудно. Или математическая задача может быть трудной лишь в некоторых

случаях, но не в среднем; так часто бывает, когда задача новая и еще плохо изучена. Так произошло с предложенной в 1978 году Мерклом и Хеллманом криптосистемой на основе задачи о рюкзаке, которая впоследствии была полностью взломана методом редукции базиса решетки.

Наконец, даже если доказательство безопасности алгоритма безупречно, его реализация может оказаться криптографически нестойкой. Например, воспользовавшись утечкой информации по побочному каналу, скажем об энергопотреблении или времени выполнения, противник может что-то узнать о внутреннем устройстве алгоритма и взломать его, обойдя доказательство. Имеется также опасность неправильного применения криптографической схемы: если алгоритм слишком сложен и имеет чересчур много конфигурационных параметров, то велики шансы, что пользователь или разработчик сконфигурирует его неправильно, что может привести к полной утрате безопасности.

Эвристическая безопасность

Доказуемая безопасность – прекрасный способ получить уверенность в качестве криптографической схемы, но не для всех видов алгоритмов такие доказательства существуют. На самом деле для большинства симметричных шифров нет доказательства безопасности. Например, мы ежедневно полагаемся на алгоритм Advanced Encryption Standard (AES), чтобы безопасно шифровать данные, которыми обмениваются мобильные телефоны, ноутбуки и настольные компьютеры, но AES не является доказуемо безопасным; никто не доказал, что его так же трудно взломать, как решить какую-то хорошо известную задачу. AES не сводится к математической задаче или к другому алгоритму, потому что сам является трудной задачей.

В тех случаях, когда доказать безопасность невозможно, остается только одна причина доверять шифру – многие компетентные специалисты пытались взломать его и потерпели неудачу. Иногда это называют *эвристической безопасностью*.

Так можно ли быть точно уверенным, что шифр безопасен? Нельзя, но есть некоторая уверенность, что алгоритм не будет взломан, если сотни опытных криптоаналитиков потратили сотни часов, пытаясь взломать его, и опубликовали полученные результаты. Обычно атаки организуются на *упрощенные версии* шифра (с меньшим числом операций или меньшим числом раундов – коротких последовательностей операций, которые повторяются для лучшего перемешивания битов).

В ходе анализа нового шифра криптоаналитики сначала пытаются взломать один раунд, затем два, три – столько, сколько смогут. *Запасом безопасности* называется разность между общим числом раундов и числом успешно атакованных раундов. Если спустя несколько лет изучения запас безопасности все еще велик, то появляется уверенность, что шифр (вероятно) безопасен.

Генерирование ключей

Если вы планируете что-то зашифровать, то нужно будет сгенерировать ключи – временные «сеансовые» (типа тех, что генерируются при посещении HTTPS-сайта) или открытые на длительный срок. Напомним (см. главу 2), что секретные ключи лежат в основе криптографической безопасности и должны генерироваться случайным образом, так чтобы их нельзя было предсказать.

Например, при заходе на HTTPS-сайт ваш браузер получает открытый ключ сайта и с его помощью генерирует симметричный ключ, действующий только на время текущего сеанса, тогда как открытый ключ сайта и ассоциированный с ним закрытый ключ могут действовать на протяжении многих лет. Поэтому было бы лучше, если бы противник не смог узнать закрытый ключ. Но генерирование секретного ключа не всегда сводится к простому получению достаточного числа псевдослучайных битов. Криптографические ключи могут генерироваться одним из трех способов:

- *случайно*, с помощью генератора псевдослучайных чисел (PRNG) и при необходимости алгоритма генерирования ключей;
- из *пароля*, с помощью функции формирования ключа (key derivation function – KDF), которая преобразует указанный пользователем пароль в ключ;
- посредством *протокола совместной выработки ключа*, состоящего из серии обмена сообщениями между двумя или более сторонами, которая завершается выработкой общего ключа.

Сейчас я объясню только простейший метод: случайное генерирование.

Генерирование симметричных ключей

Симметричный ключ – это секретный ключ, известный обеим сторонам. Такие ключи генерировать проще всего. Обычно они имеют такую же длину, как обеспечиваемый ими уровень безопасности: 128-битовый ключ обеспечивает 128-битовую безопасность, и любой из 2^{128} возможных ключей ничем не хуже любого другого.

Чтобы сгенерировать симметричный n-битовый ключ с помощью криптографически стойкого PRNG, нужно просто запросить n псевдослучайных бит и использовать их в качестве ключа. Например, для генерирования случайного симметричного ключа можно воспользоваться пакетом OpenSSL; следующая команда выводит псевдослучайные биты (понятно, что на вашей машине результат будет другой):

```
$ openssl rand 16 -hex
65a4400ea649d282b855bd2e246812c6
```

Генерирование асимметричных ключей

Асимметричные ключи обычно длиннее обеспечиваемого ими уровня безопасности. Но главная проблема не в этом. Асимметричные ключи труднее генерировать, потому что недостаточно получить *n* бит от PRNG и на этом успокоиться. Асимметричный ключ – это не просто последовательность битов, а объект определенного типа, например большое число, обладающее нужными свойствами (в RSA это произведение двух простых чисел). Маловероятно, что случайная битовая строка (а стало быть, случайное число) будет обладать свойствами, необходимыми, чтобы считаться допустимым ключом.

Для генерирования асимметричного ключа псевдослучайные биты нужно подать на вход *алгоритма генерирования ключа*. По этому начальному значению, длина которого должна быть не меньше требуемого уровня безопасности, алгоритм строит закрытый ключ и соответствующий ему открытый ключ, так чтобы эта пара удовлетворяла всем обязательным критериям. Например, наивный алгоритм генерирования ключей для RSA сгенерировал бы число $n = pq$, воспользовавшись алгоритмом нахождения двух простых чисел примерно одинаковой длины. Этот алгоритм пробует случайные числа, пока какое-нибудь не окажется простым, поэтому нам понадобится алгоритм для проверки простоты числа.

Чтобы не заниматься самостоятельной реализацией алгоритма генерирования ключей, можно воспользоваться пакетом OpenSSL и сгенерировать, например, 4096-битовый закрытый ключ RSA:

```
$ openssl genrsa 4096
Generating RSA private key, 4096 bit long modulus
.....................................................................
..............................++
.........................................++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MIIJKQIBAAKCAgEA3Qgm6OjMy61YVstaGawk22A9LyMXhiQUU4N8F5QZXEef2Pjq
vTtAIA1hzpK2AJsv16INpNkYcTjNmechAJ0xHraftO6cp2pZFP85dvknsMfUoe8u
btKXZiYvJwpS0fQQ4tzlDtH45Gj8sMHcwFxTO3HSIx0XV0owfJTLMzZbSE3TDlN+
JdW8d9Xd5UVB+o9gUCI8tSfnOjF2dHlLNiOhlfT4w0Rf+G35USIyUJZtOQ0Dh8M+
--опущено--
zO/dbYtqRkMT8Ubb/0Q1IW0q8e0WnFetzkwPzAIjwZGXT0kWJu3RYj1OXbTYDr2c
xBRVC/ujoDL6O3NaqPxkWY5HJVmkyKIE5pC04RFNyaQ8+o4APyobabPMylQq5Vo5
N5L2c4mhy1/OH8fvKBRDuvCk2oZinjdoKUo8ZA5DOa4pdvIQfR+b4/4Jjsx4
-----END RSA PRIVATE KEY-----
```

Обратите внимание, что ключ записывается в определенном формате – данные в кодировке base64, обрамленные маркерами `BEGIN RSA PRIVATE KEY` и `END RSA PRIVATE KEY`. Это стандартный формат, поддерживаемый большинством систем, которые затем преобразуют это представление в последовательность байтов. Точки в начале выдачи индицируют ход выполнения, а `e is 65537 (0x10001)` – параметр, ис-

пользуемый при шифровании (напомним, что RSA зашифровывает открытый текст, вычисляя $C = P^e \bmod n$).

Защита ключей

Получив секретный ключ, вы должны хранить его в тайне, но так чтобы он был доступен в любой момент, когда понадобится. Есть три способа решения этой проблемы.

Обертывание ключа (шифрование ключа другим ключом)

Проблема в том, что второй ключ должен быть доступен, когда понадобится дешифрировать защищенный ключ. На практике второй ключ часто генерируется из пароля, указанного пользователем в момент, когда ему нужно использовать защищенный ключ. Именно так закрытые ключи обычно защищаются в протоколе Secure Shell (SSH).

Динамическое генерирование из пароля

В этом случае никакой зашифрованный файл хранить не нужно, потому что ключ генерируется непосредственно из пароля. Этот метод используется в современных системах типа miniLock. Несмотря на сравнительную простоту, он менее распространен, чем обертывание ключа, т. к. уязвим к слабым паролям. Предположим, например, что противник перехватил какое-то зашифрованное сообщение; если использовалось обертывание ключа, то противнику нужно сначала получить файл с защищенным ключом, который обычно хранится в локальной файловой системе пользователя, так что добраться до него нелегко. Но если применялось динамическое генерирование, то противник может подобрать правильный пароль из числа кандидатов и попытаться таким образом дешифрировать сообщение. Если пароль слабый, то ключ будет скомпрометирован.

Хранение ключа в аппаратном устройстве (на смарт-карте или на электронном USB-ключе)

При таком подходе ключ хранится в защищенной памяти и останется в секрете, даже если компьютер будет скомпрометирован. Это самый безопасный способ хранения ключа, но и самый дорогой и неудобный, потому что аппаратное устройство нужно носить с собой, рискуя потерять. Смарт-карты и электронные ключи обычно не требуют ввода пароля, чтобы извлечь ключ из памяти.

Примечание *Какой бы метод вы ни использовали, следите за тем, чтобы не перепутать открытый и закрытый ключи при обмене ключами и случайно не опубликовать закрытый ключ, отправив его по электронной почте или включив в исходный код (я сам находил закрытые ключи на GitHub).*

Чтобы протестировать обертывание ключа, выполните показанную ниже команду OpenSSL с аргументом `-aes128`, который означает, что ключ нужно зашифровать шифром AES-128 (AES со 128-битовым ключом):

```
$ openssl genrsa -aes128 4096
Generating RSA private key, 4096 bit long modulus
..........++
.....................................................................
.........................................++
e is 65537 (0x10001)
Enter pass phrase:
```

Запрошенная парольная фраза будет использоваться для шифрования вновь созданного ключа.

Какие возможны проблемы

Криптографическая безопасность может быть нарушена многими способами. Самый серьезный риск – ложное чувство безопасности, возникающее благодаря имеющимся доказательствам безопасности или использованию хорошо изученных протоколов. Я продемонстрирую это на двух примерах.

Ложное чувство безопасности

Даже доказательства безопасности, предложенные известными учеными, могут содержать ошибки. Один из самых ярких примеров чудовищной ошибки в доказательстве – метод *оптимального асимметричного шифрования с дополнением* (Optimal Asymmetric Encryption Padding – OAEP), который использовал RSA и был реализован во многих приложениях. Некорректное доказательство безопасности OAEP против атаки с подобранным шифртекстом считалось правильным в течение семи лет, пока другой исследователь не обнаружил в нем дефект в 2001 году. Мало того что доказательство содержало ошибку, так еще и сам результат оказался неверен. Найденное впоследствии новое доказательство продемонстрировало, что OAEP только почти безопасен против атаки с подобранным шифртекстом. Теперь нам остается доверять новому доказательству и надеяться, что в нем нет ошибки. (Дополнительные сведения см. в статье Victor Shoup «OAEP Reconsidered», опубликованной в 2001 году.)

Короткие ключи для поддержки унаследованных приложений

В 2015 году исследователи обнаружили, что некоторые HTTPS-сайты и SSH-серверы поддерживают криптографию с открытым ключом

меньшей длины, чем ожидалось, а именно 512 бит вместе 2048. Напомним, что в схемах с открытым ключом уровень безопасности не равен длине ключа, а в случае HTTPS ключи длиной 512 бит обеспечивают уровень безопасности примерно 60 бит. Для взлома таких ключей достаточно примерно двух недель вычислений на кластере из 72 процессоров. Проблема была обнаружена на многих сайтах, в т. ч. на сайте ФБР. Хотя в конечном итоге дефект был устранен (благодаря исправлениям в OpenSSL и других программах), сюрприз был крайне неприятным.

Для дополнительного чтения

Если хотите больше узнать о доказуемой безопасности симметричных шифров, почитайте документацию по функциям губки (*http://sponge.noekeon.org/*). Функции губки ввели в симметричную криптографию подход на основе перестановок, который описывает, как построить семейство криптографических функций с помощью всего одной перестановки.

К обязательному чтению на тему реальной стоимости атаки относятся статья Bernstein «Understanding Brute Force» 2005 года и статья Wiener «The Full Cost of Cryptanalytic Attacks» 2004 года; обе можно скачать бесплатно.

Если хотите почитать об определении уровня безопасности для ключа заданного размера, зайдите на сайт *http://www.keylength.com/*. Там объясняется, как закрытые ключи защищаются в стандартных криптографических утилитах, включая SSH, OpenSSL, GnuPG и др.

Наконец, в качестве упражнения выберите какое-нибудь приложение (например, для безопасного обмена сообщениями) и разберитесь, какие в нем используются криптосхемы, длины ключей и соответствующие уровни безопасности. Зачастую вы будете встречать удивительные несогласованности, например когда первая схема обеспечивает уровень безопасности 256 бит, а вторая только 100 бит. Вся система обычно не более безопасна, чем ее самая слабая компонента.

4

БЛОЧНЫЕ ШИФРЫ

Во время холодной войны США и Советский Союз разрабатывали собственные шифры. Правительство США создало шифр Data Encryption Standard (DES), который был принят в качестве федерального стандарта в период с 1979 по 2005 год, а КГБ разработал алгоритм ГОСТ 28147–89, который держался в секрете до 1990 года и используется по сей день[1]. В 2000 году Национальный институт стандартов и технологий США (NIST) выбрал преемника DES – разработанный в Бельгии алгоритм *Advanced Encryption Standard* (*AES*), который теперь применяется в большинстве электронных устройств. У AES, DES и ГОСТ 28147–89 есть общая черта: все это *блочные шифры*, в которых базовый алгоритм, применяемый к блокам данных, сочетается с режимом работы, т. е. способом обработки последовательности блоков данных.

[1] Стандарт ГОСТ 28147–89 отменен на территории России и стран СНГ с 31 мая 2019 года и заменен стандартами ГОСТ 34.12–2018 и ГОСТ 34.13–2018. – *Прим. перев.*

В этой главе мы дадим обзор базовых алгоритмов, лежащих в основе блочных шифров, обсудим их режимы работы и объясним, как они сочетаются. Также мы рассмотрим, как работает шифр AES, и завершим главу описанием классической атаки из 1970-х годов – встреча посередине, а также любимой техники атак из 2000-х годов – оракулы дополнения.

Что такое блочный шифр?

Блочный шифр состоит из алгоритма шифрования и алгоритма дешифрирования.

- *Алгоритм шифрования* (**E**) принимает ключ K и блок открытого текста P и порождает блок шифртекста C. Операция шифрования записывается в виде $C = \mathbf{E}(K, P)$.
- *Алгоритм дешифрирования* (**D**) является обратным к алгоритму шифрования, т. е. восстанавливает исходный открытый текст P. Эта операция записывается в виде $P = \mathbf{D}(K, C)$.

Поскольку алгоритмы шифрования и дешифрирования взаимно обратны, обычно они включают похожие операции.

Цели безопасности

Если вы внимательно следили за обсуждением шифрования, случайности и неразличимости в предыдущих главах, то определение безопасного блочного шифра не вызовет удивления. Как и раньше, мы определим безопасность как подобие случайности.

Чтобы блочный шифр был безопасным, он должен являться *псевдослучайной перестановкой* (pseudorandom permutation – PRP), т. е. при условии, что ключ хранится в секрете, противник не должен иметь возможности вычислить результат применения блочного шифра к любому входу. Иными словами, коль скоро с точки зрения противника K неизвестен и случаен, он ничего не может сказать о том, как выглядит $\mathbf{E}(K, P)$ для любого заданного P.

Вообще, противник не должен иметь возможности выявить хоть какую-нибудь закономерность в связи входных и выходных данных блочного шифра. Иначе говоря, невозможно отличить блочный шифр от истинно случайной перестановки, имея доступ типа черного ящика к функциям шифрования и дешифрирования для фиксированного, но неизвестного ключа. И кроме того, противник не должен иметь возможность определить секретный ключ безопасного блочного шифра, в противном случае он мог бы воспользоваться этим ключом, чтобы отличить блочный шифр от случайной перестановки. Конечно, отсюда также следует, что противник не может предсказать открытый текст, соответствующий заданному шифртексту, порожденному блочным шифром.

Размер блока

Блочный шифр характеризуется двумя значениями: размер блока и размер ключа. Безопасность зависит от того и другого. У большинства блочных шифров блоки имеют размер 64 или 128 бит – в DES используются 64-битовые (2^6) блоки, а в AES – 128-битовые (2^7). Когда длина является степенью двойки, то упрощаются обработка, хранение и адресация данных. Но почему 2^6 и 2^7, а не, скажем, 2^4 или 2^{16} бит?

Во-первых, блок не должен быть слишком большим, чтобы минимизировать длину шифртекста и потребление памяти. Что касается длины шифртекста, то блочные шифры обрабатывают не биты, а блоки. Это значит, что для шифрования 16-битового сообщения с блоком 128 бит нужно сначала преобразовать сообщение в 128-битовый блок, и только потом блочный шифр сможет обработать его и вернуть 128-битовый шифртекст. Чем шире блоки, тем больше эти накладные расходы. Что касается *потребления памяти*, то для обработки 128-битового блока необходимо по меньшей мере 128 бит памяти. Это достаточно мало, чтобы можно было поместить данные в регистры большинства процессоров или реализовать алгоритм с помощью специализированного оборудования. Блоки длиной 64, 128 и даже 512 бит допускают эффективную реализацию в большинстве случаев. Но для блоков большего размера (например, несколько килобайтов) стоимость и производительность реализации заметно пострадают.

Если длина шифртекста или потребление памяти критичны, то можно использовать 64-битовые блоки, потому что в этом случае шифртексты будут короче, а потребление памяти меньше. В противном случае лучше выбирать 128-битовые или более длинные блоки, главным образом потому, что на современных CPU 128-битовые блоки можно обработать эффективнее, чем 64-битовые, и они обеспечивают большую безопасность. В частности, в процессорах имеются специальные команды для параллельной обработки одного или нескольких 128-битовых блоков, например семейство команд Advanced Vector Extensions (AVX) в процессорах Intel.

Атака по кодовой книге

Итак, блоки не должны быть слишком большими, но они не должны быть и слишком маленькими, иначе шифр уязвим для *атак по кодовой книге*; такие атаки против блочных шифров эффективны только при использовании небольших блоков. Для 16-битовых блоков атака по кодовой книге работает следующим образом.

1. Получить 65 536 (2^{16}) шифртекстов, соответствующих каждому из 16-битовых блоков открытого текста.
2. Построить справочную таблицу – *кодовую книгу*, – отображающую каждый блок шифртекста на соответствующий ему блок открытого текста.

3. Для дешифрирования блока неизвестного шифртекста найти соответствующий блок открытого текста в таблице.

При использовании 16-битовых блоков для справочной таблицы нужно всего $2^{16} \times 16 = 2^{20}$ бит памяти, т. е. 128 КБ. Для 32-битовых блоков объем потребной памяти возрастает до 16 ГБ, это еще осуществимо. Но для 64-битовых блоков придется хранить 2^{70} бит (один зетабит, или 128 экзабайт) – даже не думайте об этом. Для больших блоков атаки по кодовой книге не представляют проблемы.

Как устроены блочные шифры

Существуют сотни блочных шифров, но совсем немного способов их построения. Во-первых, используемые на практике блочные шифры – это не гигантские монолитные алгоритмы, а повторение *раундов* – коротких последовательностей операций, сила которых не в них самих, а в их числе. Во-вторых, есть две основные техники построения раунда: подстановочно-перестановочные сети (как в AES) и схемы Фейстеля (как в DES). В этом разделе мы обсудим, как они работают, но сначала рассмотрим атаку, которая приводит к успеху, если все раунды идентичны.

Раунды блочного шифра

Вычисление блочного шифра сводится к вычислению последовательности *раундов*. Раундом называется базовое преобразование, которое легко описать и реализовать и которое повторяется несколько раз, образуя алгоритм шифра. Такую конструкцию – небольшой компонент, повторяемый многократно, – проще реализовать и проанализировать, чем гигантский монолитный алгоритм.

Например, блочный шифр с тремя раундами шифрует открытый текст P путем вычисления $C = \mathbf{R}_3(\mathbf{R}_2(\mathbf{R}_1(P)))$, где $\mathbf{R}_1$, $\mathbf{R}_2$ и $\mathbf{R}_3$ – раунды. Для каждого раунда должно существовать обращение, чтобы получатель мог восстановить открытый текст. Точнее, $P = \mathbf{iR}_1(\mathbf{iR}_2(\mathbf{iR}_3(C)))$, где $\mathbf{iR}_1$ – преобразование, обратное $\mathbf{R}_1$, и так далее.

Функции раундов – $\mathbf{R}_1$, $\mathbf{R}_2$ и т. д. – обычно представляют собой одинаковые алгоритмы, параметризованные *ключом раунда*. Две функции раундов с разными ключами ведут себя по-разному, поэтому порождают разные выходы для одного и того же входа.

Ключи раундов выводятся из главного ключа K с помощью алгоритма *развертки ключа*. Например, $\mathbf{R}_1$ принимает ключ раунда K_1, $\mathbf{R}_2$ – ключ раунда K_2 и т. д.

Ключи всех раундов должны различаться. Поэтому не все ключи раунда равны ключу K, иначе раунды были бы идентичны и блочный шифр оказался бы менее безопасным, как демонстрируется в следующем разделе.

Сдвиговая атака и ключи раунда

В блочном шифре никакой раунд не должен совпадать с другим раундом, чтобы избежать *сдвиговой атаки*. При сдвиговой атаке ищутся две пары «открытый текст – шифртекст», (P_1, C_1) и (P_2, C_2), где $P_2 = \mathbf{R}(P_1)$, а **R** – раунд шифра (см. рис. 4.1). Если раунды идентичны, то из связи между двумя открытыми текстами, $P_2 = \mathbf{R}(P_1)$, вытекает связь $C_2 = \mathbf{R}(C_1)$ между соответствующими шифртекстами. На рис. 4.1 показано три раунда, но связь $C_2 = \mathbf{R}(C_1)$ будет иметь место вне зависимости от числа раундов, будь то 3, 10 или 100. Проблема в том, что знание входа и выхода одного раунда часто позволяет найти ключ. (Детали см. в статье Biryukov and Wagner «Advanced Slide Attacks», опубликованной в 1999 году и доступной по адресу *https://www.iacr.org/archive/eurocrypt2000/1807/18070595-new.pdf*.)

Использование ключей раунда в качестве параметров гарантирует, что раунды будут вести себя по-разному, и потому расстраивает сдвиговые атаки.

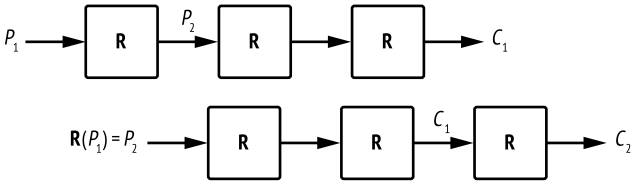

Рис. 4.1. Принцип сдвиговой атаки против блочных шифров с идентичными раундами

Примечание *Потенциальный побочный эффект и достоинство использования ключей раундов – защита от атак по побочному каналу, т. е. атак, в которых используется информация, просачивающаяся наружу из-за особенностей реализации шифра (например, электромагнитное излучение). Если преобразование главного ключа K в ключ раунда K_i необратимо, то противник, сумевший определить K_i, не сможет использовать эту информацию для нахождения K. К сожалению, лишь в немногих блочных шифрах используется односторонняя развертка ключа. Так, алгоритм развертки ключа в AES позволяет противнику вычислить K по ключу любого раунда K_i.*

Подстановочно-перестановочные сети

Если вы читали книги по криптографии, то наверняка встречали термины *конфузия* и *диффузия*. Конфузия означает, что входные данные (открытый текст и ключ шифрования) подвергаются сложным преобразованиям, а диффузия – что эти преобразования в равной мере зависят от всех битов входных данных. На верхнем уровне конфузия описывает глубину, а диффузия – ширину преобразования. При проектировании блочного шифра конфузия и диффузия принимают вид операций подстановки и перестановки, которые объединяются

в подстановочно-перестановочные сети (substitution–permutation network – SPN).

Подстановка часто встречается в форме *S-блоков*, или *подстановочных блоков*, – небольших таблиц перекодировки для преобразования блоков длиной 4 или 8 бит. Например, первый из восьми S-блоков шифра Serpent состоит из 16 элементов (3 8 f 1 a 6 5 b e d 4 2 7 0 9 c), где каждый элемент представляет 4-битовый полубайт. Этот конкретный S-блок отображает полубайт 0000 в 3 (0011), полубайт 0101 (десятичное 5) в 6 (0110) и т. д.

Примечание *S-блоки следует тщательно выбирать, обращая внимание на криптографическую стойкость: они должны быть максимально нелинейны (входы и выходы должны быть связаны сложными уравнениями) и не иметь статистического смещения (это означает, например, что изменение одного входного бита потенциально может повлиять на любой из выходных битов).*

Перестановка в подстановочно-перестановочной сети может быть совсем простой, например изменение порядка битов, – такую перестановку легко реализовать, но она не очень хорошо перемешивает биты. В некоторых шифрах вместо переупорядочения битов применяется умножение на матрицы, т. е. серия операций умножения на фиксированные значения (элементы матрицы) с последующим сложением результатов. Такие линейно-алгебраические операции могут создавать зависимости между всеми битами, т. е. обеспечивают сильную диффузию. Например, в блочном шифре FOX 4-байтовый вектор (a, b, c, d) преобразуется в (a', b', c', d') следующим образом:

$$a' = a + b + c + (2 \times d);$$
$$b' = a + (253 \times b) + (2 \times c) + d;$$
$$c' = (253 \times a) + (2 \times b) + c + d;$$
$$d' = (2 \times a) + b + (253 \times c) + d.$$

В этих формулах числа 2 и 253 интерпретируются как двоичные полиномы, а не целые числа, поэтому операции сложения и умножения определены несколько иначе, чем мы привыкли. Например, вместо 2 + 2 = 4 имеем 2 + 2 = 0. Но главное то, что каждый байт начального состояния оказывает влияние на все 4 байта конечного.

Схемы Фейстеля

В 1970-х годах сотрудник компании IBM Хорст Фейстель спроектировал блочный шифр Lucifer, работающий следующим образом:

1. Разделить 64-битовый блок на две половины по 32 бита, L и R.
2. Положить L равным $L \oplus \mathbf{F}(R)$, где **F** – подстановочно-перестановочный раунд.
3. Обменять местами L и R.

4. Перейти к шагу 2 и повторить 15 раз.
5. Объединить L и R в 64-битовый выходной блок.

Это построение, получившее название *схема Фейстеля*, показано на рис. 4.2. Слева показана только что описанная схема, а справа – функционально эквивалентное представление, где вместо обмена L и R чередуются операции $L = L \oplus \mathbf{F}(R)$ и $R = R \oplus \mathbf{F}(L)$.

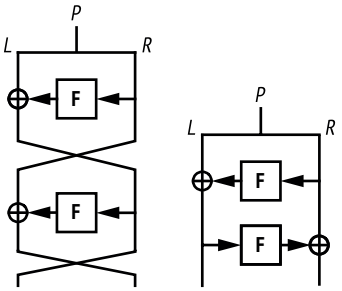

Рис. 4.2. Две эквивалентные формы построения блочного шифра со схемой Фейстеля

На рис. 4.2 ключи опущены для упрощения диаграмм, но отметим, что первому вызову **F** передается ключ первого раунда K_1, а второму – ключ другого раунда K_2. В DES функция **F** принимает 48-битовый ключ, который выводится из 56-битового ключа K.

В схеме Фейстеля функция **F** может быть псевдослучайной перестановкой (pseudorandom permutation – PRP) или псевдослучайной функцией (pseudorandom function – PRF). PRP дает разные выходы для разных входов, а в случае PRF существуют такие X и Y, что $\mathbf{F}(X) = \mathbf{F}(Y)$. Но в схеме Фейстеля это различие несущественно, коль скоро **F** криптографически стойкая.

Сколько раундов должно быть в схеме Фейстеля? В DES таких раундов 16, в ГОСТ 28147–89 – 32. Если функция **F** настолько криптографически стойкая, насколько возможно, то теоретически хватает четырех раундов, но в реальных шифрах раундов больше, чтобы противостоять возможным слабостям в **F**.

Шифр Advanced Encryption Standard (AES)

AES – самый распространенный шифр на земле. До принятия AES стандартным считался шифр DES, со своей до смешного низкой 56-битовой безопасностью, а также усовершенствованный вариант DES – Triple DES, или 3DES.

Хотя 3DES обеспечивает более высокий уровень безопасности (112 бит), этого недостаточно, потому что для достижения 112-битовой безопасности длина ключа должна составлять 168 бит, а программная реализация алгоритма работает медленно (DES создавался для быстрой реализации в интегральных схемах, а не на универсальных процессорах). AES исправляет оба недостатка.

NIST стандартизовал AES в 2000 году как замену DES, после чего этот шифр стал всемирным стандартом шифрования де-факто. Большинство современных продуктов для шифрования поддерживают AES, а АНБ одобрило его для защиты совершенно секретной информации. (Некоторые страны предпочитают использовать собственные шифры в основном потому, что не желают пользоваться американским стандартом, но корни AES скорее бельгийские, чем американские.)

Примечание *AES раньше назывался Rijndael (стяжение имен авторов, Rijmen и Daemen, произносится «рейндал»). Это был один из 15 кандидатов в конкурсе на усовершенствованный стандарт шифрования, который проводился NIST с 1997 по 2000 год и итогом которого должен был стать «незасекреченный, открытый всем интересующимся алгоритм шифрования, способный защитить секретную государственную информацию в следующем веке» – так формулировалась цель объявленного конкурса в Федеральном регистре. Конкурс AES был своего рода «смотром талантов» для криптографов, в котором мог участвовать любой желающий – либо предложив свой шифр, либо взломав шифр, предложенный другими участниками.*

Внутреннее устройство AES

AES обрабатывает 128-битовые блоки, используя секретный ключ длиной 128, 192 или 256 бит. 128-битовый ключ наиболее распространен, потому что шифрование при этом производится чуть быстрее, а разница между 128- и 256-битовой безопасностью для большинства приложений несущественна.

В то время как некоторые шифры работают с отдельными битами или 64-битовыми словами, AES манипулирует *байтами*. 16-байтовый открытый текст рассматривается как двумерный массив байтов ($s = s_0$, s_1, ..., s_{15}), как показано на рис. 4.3. (Буква s используется, потому что этот массив называется *внутренним состоянием* (state), или просто *состоянием*.) AES преобразует байты, столбцы и строки этого массива, порождая шифртекст.

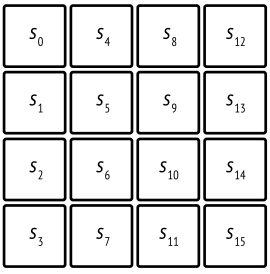

Рис. 4.3. Внутреннее состояние AES, представленное в виде массива 16 байт размером 4×4

Для преобразования состояния в AES используется подстановочно-перестановочная сеть SPN, показанная на рис. 4.4, с 10 раундами для 128-битовых ключей, 12 – для 192-битовых и 14 – для 256-битовых.

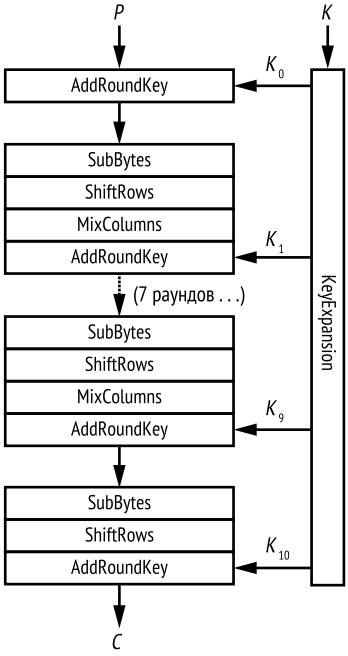

Рис. 4.4. Внутренние операции AES

На рис. 4.4 показаны все четыре строительных блока, составляющих один раунд AES (заметьте, что все раунды, кроме последнего, являются последовательностями операций SubBytes, ShiftRows, MixColumns и AddRoundKey).

- **AddRoundKey.** Применяет XOR к ключу раунда и внутреннему состоянию.
- **SubBytes.** Заменяет каждый байт (s_0, s_1, ..., s_{15}) другим байтом согласно S-блоку. В этом пример S-блок представляет собой таблицу перекодировки из 256 элементов.
- **ShiftRows.** Циклически сдвигает *i*-ю строку на *i* позиций, где *i* изменяется от 0 до 3 (см. рис. 4.5).
- **MixColumns.** Применяет одно и то же линейное преобразование к каждому из четырех столбцов состояния (т. е. к каждой группе ячеек, окрашенных одним и тем же оттенком серого в левой части рис. 4.5).

Напомним, что в акрониме SPN буква *S* означает «подстановка», а буква *P* – «перестановка». Здесь подстановочным слоем является SubBytes, а перестановочным – комбинация ShiftRows и MixColumns.

Функция *KeyExpansion*, показанная на рис. 4.4, – реализация алгоритма развертки ключа в AES. В результате развертки из одного 16-байтового ключа создается 11 ключей раундов ($K_0, K_1, \ldots, K_{10}$) длиной 16 байт каждый, для чего используется тот же S-блок, что в SubBytes, и комбинация операций XOR. Важное свойство KeyExpansion заключается в том, что если известен любой ключ раунда K_i, то противник может определить ключи всех остальных раундов и главный ключ K,

обратив алгоритм. Возможность получить ключ по ключу любого раунда обычно считается неидеальной защитой от атак по побочному каналу, если противник может легко определить ключ раунда.

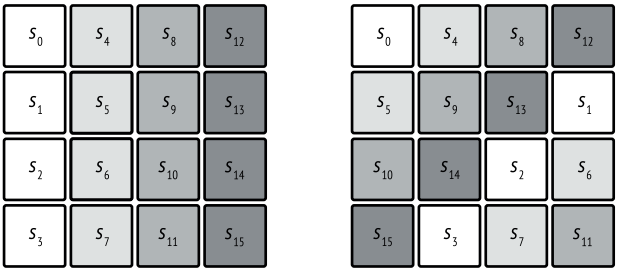

Рис. 4.5. ShiftRows циклически сдвигает байты в каждой строке внутреннего состояния

Без этих операций AES был бы абсолютно небезопасен. Каждая операция вносит свой вклад в безопасность AES.

- Без KeyExpansion во всех раундах использовался бы один и тот ключ *K*, и AES был бы уязвим к сдвиговым атакам.
- Без AddRoundKey шифрование не зависело бы от ключа, и тогда любой мог бы дешифрировать шифртекст, не зная ключа.
- SubBytes привносит нелинейные операции, повышающие криптографическую стойкость. Без нее AES был бы просто большой системой линейных уравнений, решаемой школьными методами.
- Без ShiftRows изменения в любом столбце не оказывали бы влияния на другие столбцы, а это значит, что AES можно было бы взломать, построив четыре кодовые книги по 2^{32} элемента, по одной для каждого столбца. (Напомним, что в безопасном блочном шифре изменение одного бита входных данных должно влиять на все выходные биты.)
- Без MixColumns изменение одного байта не влияло бы на остальные байты состояния. Тогда атака с подобранным открытым текстом позволила бы дешифрировать любой шифртекст после сохранения 16 справочных таблиц по 256 байт каждая, в которых хранятся зашифрованные значения каждого возможного значения байта.

Заметим, что последний раунд AES на рис. 4.4 не включает операции MixColumns. Она опущена, чтобы исключить бесполезные вычисления: поскольку MixColumns – линейная (а значит, предсказуемая) операция, ее воздействие в самом последнем раунде можно аннулировать, скомбинировав биты способом, не зависящим ни от их значения, ни от ключа. Однако SubBytes невозможно инвертировать, не зная значения состояния в момент, предшествующий AddRoundKey.

Для дешифрирования шифртекста AES заменяет каждую операцию обратной к ней: для обращения SubBytes нужно взять обратную справочную таблицу, для обращения ShiftRow произвести циклический

сдвиг в противоположном направлении, для обращения MixColumns применить преобразование, описываемое обратной матрицей, а AddRoundKey не изменяется, потому что обращением операции XOR является она сама.

AES в действии

Чтобы потренироваться в зашифровывании и дешифрировании с помощью AES, можете воспользоваться криптографической библиотекой Python, как в листинге 4.1.

Листинг 4.1. Эксперименты с AES с применением криптографической библиотеки Python

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

# выбрать случайный 16-байтовый ключ, воспользовавшись PRNG, включенным в Python
k = urandom(16)
print “k = %s” % hexa(k)
# создать экземпляр AES-128 для шифрования одного блока
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend = default_backend())
aes_encrypt = cipher.encryptor()

# выбрать в качестве блока открытого текста строку, состоящую из одних нулей
p = ‘\x00’*16
# зашифровать открытый текст p, получив шифртекст c
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print “enc(%s) = %s” % (hexa(p), hexa(c))
# дешифрировать шифртекст c в открытый текст p
aes_decrypt = cipher.decryptor()
p = aes_decrypt.update(c) + aes_decrypt.finalize()
print “dec(%s) = %s” % (hexa(c), hexa(p))
При выполнении этого скрипта будет напечатано что-то типа:
$ ./aes_block.py
k = 2c6202f9a582668aa96d511862d8a279
enc(00000000000000000000000000000000) = 12b620bb5eddcde9a07523e59292a6d7
dec(12b620bb5eddcde9a07523e59292a6d7) = 00000000000000000000000000000000
```

На вашем компьютере результат будет иным, потому что ключ выбирается случайно при каждом выполнении.

Реализация AES

Настоящие реализации AES работают не так, как алгоритм на рис. 4.4. В производственном коде AES вы не увидите вызова функции Sub-

Bytes(), за ней ShiftRows(), а за ней MixColumns(), потому что это было бы неэффективно. Вместо этого в быстрых программных реализациях AES используется специальная техника, называемая табличной реализацией, и машинные команды.

Табличные реализации

В табличных реализациях AES последовательность операций SubBytes-ShiftRows-MixColumns заменяется комбинацией XOR и поиска в таблицах, зашитых в код программы и загружаемых в память во время выполнения. Это возможно, потому что MixColumns эквивалентна XOR четырех 32-битовых значений, каждое из которых зависит от одного байта состояния и от SubBytes. Таким образом, можно построить четыре таблицы по 256 элементов (по одному для каждого возможного значения байта) и реализовать последовательность SubBytes-MixColumns с помощью выборки четырех 32-битовых значений из этих таблиц и применения к ним XOR.

Например, табличная реализация на C в пакете OpenSSL показана в листинге 4.2.

Листинг 4.2. Табличная реализация AES на C в OpenSSL

```
/* раунд 1: */
t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
/* раунд 2: */
s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
--опущено--
```

Для базовой табличной реализации шифрования AES необходимо 4 килобайта, поскольку в каждой таблице хранится 256 32-битовых значений, занимающих $256 \times 32 = 8192$ бита, или один килобайт. Для дешифрирования нужно еще четыре таблицы и, стало быть, еще четыре килобайта. Но существуют приемы, позволяющие уменьшить объем памяти с четырех до одного килобайта и даже меньше.

К сожалению, табличные реализации уязвимы к *атакам с хронометражем кеша*, в которых используются вариации во времени чтения или записи элементов кеш-памяти. Время доступа к элементу зависит от его относительной позиции в кеше. Поэтому хронометраж дает информацию о том, к какому элементу производилось обращение, а значит, и информацию о секретной части шифра.

От атак с хронометражем кеша трудно уклониться. Очевидное решение – вообще отказаться от справочных таблиц, написав програм-

му, время выполнения которой не зависит от входных данных, но это почти невозможно сделать с сохранением быстродействия, поэтому производители микросхем выбрали радикальное решение: вместо того чтобы полагаться на потенциально уязвимое программное обеспечение, они полагаются на *оборудование*.

Машинные команды

Машинные команды AES (AES-NI) решают проблему атак с хронометражем кеша, присущую программным реализациям AES. Чтобы понять, как работает AES-NI, подумаем о том, как вообще программа исполняется на оборудовании: для выполнения программы микропроцессор транслирует двоичный код в последовательность команд, исполняемых компонентами интегральной схемы. Например, ассемблерная команда `MUL` умножения двух 32-разрядных значений активирует транзисторы, реализующие 32-разрядный умножитель в микропроцессоре. Для реализации криптографического алгоритма мы обычно просто записываем комбинацию таких элементарных операций – сложения, умножения, XOR и т. д., – а микропроцессор активирует свои цепи сложения, умножения и XOR в предписанном порядке.

Машинные команды AES переводят эту идею на новый уровень, предоставляя разработчикам специализированные ассемблерные команды для вычисления AES. Вместо того чтобы записывать раунд AES в виде последовательности обычных ассемблерных команд, разработчик просто использует команду `AESENC`, а микросхема вычисляет раунд, т. е. вся работа перекладывается на процессор.

Типичный ассемблерный код AES с применением машинных команд показан в листинге 4.3.

Листинг 4.3. Машинные команды AES

```
PXOR       %xmm5, %xmm0
AESENC     %xmm6, %xmm0
AESENC     %xmm7, %xmm0
AESENC     %xmm8, %xmm0
AESENC     %xmm9, %xmm0
AESENC     %xmm10, %xmm0
AESENC     %xmm11, %xmm0
AESENC     %xmm12, %xmm0
AESENC     %xmm13, %xmm0
AESENC     %xmm14, %xmm0
AESENCLAST %xmm15, %xmm0
```

Этот код зашифровывает 128-битовый открытый текст, первоначально помещенный в регистр `xmm0`, в предположении, что в регистрах от `xmm5` до `xmm15` хранятся предварительно вычисленные ключи раундов, а каждая команда помещает свой результат в `xmm0`. Первая команда `PXOR` выполняет XOR с первым ключом раунда до вычисле-

ния первого раунда, а последняя команда AESENCLAST выполняет последний раунд, который немного отличается от всех остальных (операция MixColumns опущена).

Примечание *AES работает примерно в 10 раз быстрее на платформах, где реализованы машинные команды. На момент написания книги к таковым относятся практически все ноутбуки, настольные компьютеры и серверы, а также большинство мобильных телефонов и планшетов. На самом деле в последней микроархитектуре Intel команда AESENC имеет задержку четыре такта, а обратная пропускная способность составляет один такт; это означает, что для завершения AESENC требуется четыре такта, а начинать новую команду можно в каждом такте. Таким образом, для последовательного шифрования серии блоков требуется 4 × 10 = 40 тактов, чтобы выполнить 10 раундов, или 40/16 = 2,5 такта на один байт. При тактовой частоте 2 ГГц (2×10^9 тактов в секунду) мы получаем пропускную способность 736 МБ/с. Если блоки шифруются или дешифрируются независимо друг от друга, как в некоторых режимах работы, то четыре блока можно обрабатывать параллельно, полностью задействовав возможности цепей AESENC; тогда задержка составит 10 тактов на блок вместо 40, т. е. эффективная пропускная способность будет равна приблизительно 3 ГБ/с.*

Безопасен ли AES?

AES настолько безопасен, насколько это возможно для блочного шифра, и взломать его никогда не удастся. Главная причина безопасности AES заключается в том, что все выходные биты зависят от всех входных битов сложным псевдослучайным образом. Чтобы добиться этого, проектировщики AES тщательно выбирали каждый компонент, преследуя конкретные цели – MixColumns ради свойств максимальной диффузии, а SubBytes для достижения оптимальной нелинейности – и показали, что эта композиция защищает AES от целых классов криптоаналитических атак.

Но не существует доказательства того, что AES неуязвим для всех возможных атак. Прежде всего мы не знаем, какие вообще атаки возможны, и не всегда умеем доказывать, что шифр безопасен относительно конкретной атаки. Единственный способ обрести уверенность в безопасности AES – коллективные усилия: когда много квалифицированных специалистов пытаются взломать AES и, хочется надеяться, терпят неудачу.

По прошествии 15 лет и сотен публикаций о теоретической безопасности AES почти ничего неизвестно. В 2011 году криптоаналитики нашли способ восстановить ключ AES-128, выполнив 2^{126} операций вместо 2^{128}, добившись тем самым четырехкратного ускорения. Но эта «атака» требует безумного количества пар «открытый текст – шифртекст» – общим объемом примерно 2^{88} бит. Иными словами, открытие любопытное, но не из тех, которые заставят вас нервничать.

Резюмируя, можно сказать, что при реализации и развертывании криптосистем есть миллион вещей, о которых нужно побеспокоиться, но безопасность AES не из их числа. Самая серьезная опасность, грозящая блочным шифрам, – не их базовые алгоритмы, а режимы работы. Если выбран неправильный режим работы или правильный режим неправильно используется, то даже такой стойкий шифр, как AES, вас не спасет.

Режимы работы

В главе 1 я объяснял, как схемы шифрования сочетают перестановку с режимом работы, чтобы можно было обрабатывать сообщения любой длины. В этом разделе я рассмотрю основные режимы работы блочных шифров, их свойства (функциональные и безопасности) и когда следует (или не следует) использовать каждый. Начну с самого простого режима: электронной кодовой книги.

Режим электронной кодовой книги (ECB)

Простейший режим работы блочного шифра – это режим электронной кодовой книги (electronic codebook – ECB)[1], его и режимом-то работы назвать трудно. ECB принимает блоки открытого текста $P_1, P_2, \ldots, P_N$ и обрабатывает их независимо, вычисляя $C_1 = \mathbf{E}(K, P_1)$, $C_2 = \mathbf{E}(K, P_2)$ и т. д., как показано на рис. 4.6. Это простая, но и небезопасная операция. Повторяю: режим ECB небезопасен, использовать его не следует!

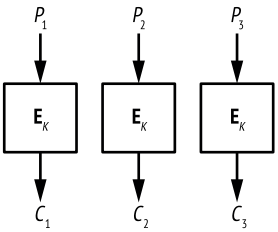

Рис. 4.6. Режим ECB

Марш Рэй, криптограф из Microsoft, писал: «Все знают, что режим ECB никуда не годится, потому что мы можем видеть пингвина». Он имел в виду знаменитую иллюстрацию небезопасности ECB, в которой используется изображение символа Linux, пингвина Такса, показанное на рис. 4.7. Слева приведено оригинальное изображение Такса, а справа оно же, зашифрованное AES в режиме ECB (впрочем, базовый шифр не имеет значения). В зашифрованном изображении легко угадывается силуэт пингвина, поскольку все блоки исходного изображения одного оттенка серого после шифрования тоже имеют один оттенок; иными словами, шифрование в режиме ECB дает то же изображение, только в других цветах.

[1] В ГОСТ 28147–89 он называется *режимом простой замены*. – *Прим. перев.*

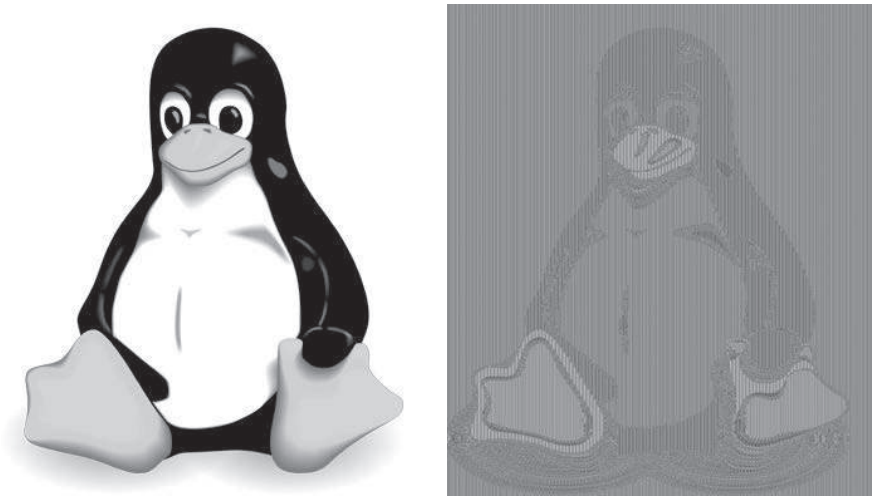

Рис. 4.7. Исходное изображение (слева) и оно же, зашифрованное в режиме ECB (справа)

Программа на Python в листинге 4.4 также демонстрирует небезопасность ECB. Она выбирает псевдослучайный ключ и шифрует 32-байтовое сообщение p, содержащее два блока нулевых байтов. Заметьте, что шифрование дает два идентичных блока, а при повторном шифровании с тем же ключом и тем же открытым текстом снова получаются те же самые два блока.

Листинг 4.4. Использование AES в режиме ECB в Python

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ‘ ‘.join(split)

k = urandom(16)
print “k = %s” % hexa(k)

# создать экземпляр AES-128 для шифрования и дешифрирования
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend=default_backend())
aes_encrypt = cipher.encryptor()

# блок p открытого текста содержит все нули
p = ‘\x00’*BLOCKLEN*2

# зашифровать открытый текст p в шифртекст c
```

```
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print “enc(%s) = %s” % (blocks(p), blocks(c))
```

В результате выполнения этого скрипта мы получим блоки шифртекста, например:

```
$ ./aes_ecb.py
k = 50a0ebeff8001250e87d31d72a86e46d
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
5eb4b7af094ef7aca472bbd3cd72f1ed 5eb4b7af094ef7aca472bbd3cd72f1ed
```

Как видим, в режиме ECB одинаковые блоки шифртекста являются для противника указанием на то, что блоки открытого текста одинаковы, – не важно, являются ли эти блоки частями одного шифртекста или разных. Это говорит о том, что блочные шифры в режиме ECB семантически небезопасны.

Еще одна проблема, связанная с ECB, заключается в том, что принимают только полные блоки данных, поэтому если длина блока составляет 16 байт, как в AES, то зашифровать можно будет сообщения размера 16 байт, 32 байта, 48 байт и вообще любого размера, кратного 16. Эту проблему можно решить несколькими способами, как мы увидим при обсуждении следующего режима, CBC. (Не буду рассказывать, как эти приемы работают с ECB, потому что использовать данный режим вообще не следует.)

Режим сцепления блоков шифртекста (CBC)

Режим сцепления блоков шифртекста (CBC) похож на ECB, но с одним маленьким отличием, которое всё меняет: результатом шифрования i-го блока, P_i, является не $C_i = \mathbf{E}(K, P_i)$, а $C_i = \mathbf{E}(K, P_i \oplus C_{i-1})$, где C_{i-1} – предыдущий блок шифртекста, т. е. блоки C_{i-1} и C_i *сцепляются*. Для первого блока P_1 предыдущего блока шифртекста не существует, поэтому берется случайное начальное значение (initial value – IV), как показано на рис. 4.8.

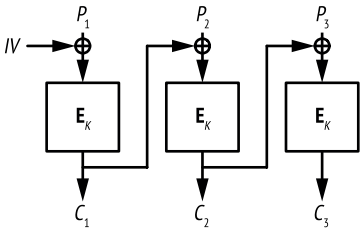

Рис. 4.8. Режим CBC

В режиме CBC каждый блок шифртекста зависит от всех предыдущих блоков и гарантируется, что одинаковым блокам открытого текста не будут соответствовать одинаковые блоки шифртекста. А случайный выбор начального значения гарантирует, что при повторном

шифровании одинаковых открытых текстов будут получаться разные шифртексты.

В листинге 4.5 иллюстрируются оба этих преимущества. Программа принимает 32-байтовое сообщение, состоящее из одних нулей (как в листинге 4.4), дважды зашифровывает его в режиме CBC и выводит оба шифртекста. В строке `iv = urandom(16)`, выделенной полужирным шрифтом, выбирается случайное начальное значение для каждого нового шифрования.

Листинг 4.5. Использование AES в режиме CBC

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
# функция blocks() разбивает строку данных на блоки, разделенные пробелами
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ‘ ‘.join(split)
k = urandom(16)
print “k = %s” % hexa(k)
# выбрать случайное начальное значение IV
iv = urandom(16)
print “iv = %s” % hexa(iv)
# создать экземпляр AES в режиме CBC
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()

p = ‘\x00’*BLOCKLEN*2
c = aes.update(p) + aes.finalize()
print “enc(%s) = %s” % (blocks(p), blocks(c))
# теперь с другим IV и тем же самым ключом
iv = urandom(16)
print “iv = %s” % hexa(iv)
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()
c = aes.update(p) + aes.finalize()
print “enc(%s) = %s” % (blocks(p), blocks(c))
```

Оба открытых текста одинаковы (состоят из одних нулей), но зашифрованные блоки должны быть разными, как в примере ниже:

```
$ ./aes_cbc.py
k = 9cf0d31ad2df24f3cbbefc1e6933c872
iv = 0a75c4283b4539c094fc262aff0d17af
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
370404dcab6e9ecbc3d24ca5573d2920 3b9e5d70e597db225609541f6ae9804a
iv = a6016a6698c3996be13e8739d9e793e2
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
655e1bb3e74ee8cf9ec1540afd8b2204 b59db5ac28de43b25612dfd6f031087a
```

К сожалению, режим CBC часто применяется с постоянным IV вместо случайного, поэтому становятся видны одинаковые открытые тексты и открытые тексты, начинающиеся с одинаковых блоков. Предположим, к примеру, что двухблочный открытый текст $P_1 || P_2$ зашифровывается в режиме CBC в двухблочный шифртекст $C_1 || C_2$. Если $P_1 || P'_2$ зашифрован с таким же IV, где P'_2 – блок, отличный от P_2, то шифртекст будет иметь вид $C_1 || C'_2$, где C'_2 отличен от C_2, но первые блоки совпадают. Таким образом, противник может догадаться, что первые блоки обоих открытых текстов одинаковы, даже если видит только шифртексты.

Примечание *В режиме CBC при дешифрировании необходимо знать IV, поэтому IV передается вместе с шифртекстом в открытом виде.*

В режиме CBC дешифрирование можно значительно ускорить, воспользовавшись распараллеливанием. Если при шифровании очередного блока P_i необходимо дождаться предыдущего блока C_{i-1}, то при дешифрировании блока вычисляется $P_i = \mathbf{D}(K, C_i) \oplus C_{i-1}$, и необходимости в предыдущем блоке открытого текста, P_{i-1}, не возникает. Это означает, что все блоки можно дешифрировать одновременно при условии, что известен предыдущий блок шифртекста, а так оно обычно и есть.

Как зашифровать любое сообщение в режиме CBC

Вернемся к проблеме последнего блока и посмотрим, как обработать открытый текст, длина которого не кратна длина блока. Например, как зашифровать 18-байтовый открытый текст шифром AES-CBC с блоками длиной 16 байт? Что делать с двумя остающимися байтами? Мы рассмотрим два широко распространенных способа решения этой проблемы. Первый, дополнение, приводит к тому, что шифртекст оказывается немного длиннее открытого текста, а второй, *заимствование шифртекста*, порождает шифртекст такой же длины, как у открытого текста.

Дополнение сообщения

Дополнение позволяет зашифровать сообщение любой длины, даже меньшее одного блока. Для блочных шифров дополнение описано в стандарте PKCS#7 и в документе RFC 5652 и используется почти всегда, когда применяется режим CBC, например при шифровании подключений по протоколу HTTPS.

Цель дополнения – расширить сообщение до полного блока, добавив в открытый текст лишние байты. Приведем правила дополнения 16-байтовых блоков.

- Если остался один байт, например если длина открытого текста равна 1, 17 или 33 байтам, то дополнить сообщение 15 байтами 0f (десятичное 15).

- Если осталось два байта, то дополнить сообщение 14 байтами 0e (десятичное 14).
- Если осталось три байта, то дополнить сообщение 13 байтами 0d (десятичное 13).

Если в блоке не хватает одного байта, то он дополняется одним байтом, равным 01. Если длина открытого текста уже кратна 16, то добавляется 16 байт 10 (десятичное 16). Ну, вы поняли идею. Этот прием обобщается на любую длину блока, не большую 255 (для более длинных блоков так не получится, потому что одним байтом нельзя закодировать значение, большее 255).

Дешифрирование дополненного сообщения производится следующим образом.

1. Дешифрировать все блоки, как в режиме CBC.
2. Проверить, удовлетворяют ли последние байты последнего блока правилу дополнения: что блок заканчивается по крайней мере одним байтом 01, или по крайней мере двумя байтами 02, или по крайней мере тремя байтами 03 и т. д. Если правило дополнения не выполнено, например если последние байты равны 01 02 03, то сообщение отвергается. В противном случае дешифратор отбрасывает байты дополнения и возвращает оставшиеся байты открытого текста.

Недостаток дополнения заключается в том, что шифртекст оказывается длиннее по крайней мере на один байт, а в худшем случае – на целый блок.

Заимствование шифртекста

Заимствование шифртекста – еще один способ шифрования сообщения, длина которого не кратна размеру блока. Этот способ сложнее и не так популярен, как дополнение, но предлагает как минимум три преимущества:

- открытый текст может иметь любую длину в *битах*, а не только в байтах. Например, можно зашифровать сообщение длиной 131 бит;
- длина шифртекста совпадает с длиной открытого текста;
- заимствование шифртекста не уязвимо к атакам на оракул дополнения, которые иногда оказываются успешны против CBC с дополнением (как мы увидим в разделе «Атаки на оракул дополнения» ниже).

В режиме CBC с заимствованием шифртекста последний неполный блок открытого текста дополняется битами из предыдущего блока шифртекста, после чего получившийся блок зашифровывается. Последний неполный блок шифртекста включает начальные биты предыдущего блока шифртекста, т. е. те, которые не были дописаны в конец последнего блока открытого текста.

На рис. 4.9 показано три блока, из которых последний, P_3, неполон (недостающие биты представлены нулем). P_3 объединяется с помощью операции XOR с последними битами предыдущего блока шифртекста, и зашифрованный результат возвращается в виде $C2$. А последний блок шифртекста, C_3, состоит из начальных битов предыдущего блока шифртекста. В процессе дешифрирования эта операция просто инвертируется.

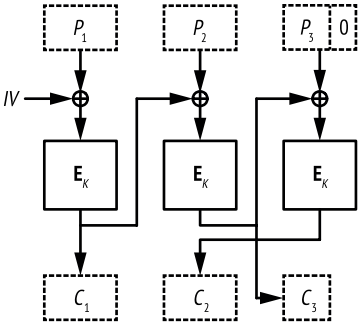

Рис. 4.9. Заимствование шифртекста в режиме шифрования CBC

У заимствования шифртекста нет никаких серьезных проблем, но оно неэлегантно и его трудно реализовать правильно, особенно учитывая, что в стандарте NIST описано три способа его реализации (см. специальную публикацию 800-38A).

Режим счетчика (CTR)

Чтобы избежать проблем, присущих заимствованию шифртекста, сохранив в то же время все его преимущества, следует использовать режим счетчика (counter mode – CTR). Режим CTR трудно назвать блочным шифром, он преобразует блочный шифр в потоковый, который принимает и отдает отдельные биты, не утруждая себя понятием блока. (Потоковые шифры подробно обсуждаются в главе 5.)

В режиме CTR (см. рис. 4.10) алгоритм блочного шифра не преобразует данные открытого текста. Вместо этого шифруются блоки, состоящие из *счетчика* и *одноразового числа* (nonce). Счетчик – это целое число, увеличивающееся на единицу для каждого блока. Никакие два блока одного сообщения не должны использовать одно и то же значение счетчика, но для разных сообщений возможны одинаковые последовательности значений счетчиков (1, 2, 3, ...). Одноразовое число, как следует из названия, можно использовать только один раз. Оно одинаково для всех блоков одного сообщения, но для разных сообщений одинаковые одноразовые числа не допускаются.

Как видно по рис. 4.10, в режиме CTR шифрование заключается в применении XOR к открытому тексту и потоку, полученному «шифрованием» одноразового числа *N* и счетчика *Ctr*. Дешифрирование производится точно так же, поэтому для шифрования и дешифрирования нужен только алгоритм шифрования. Пример на Python приведен в листинге 4.6.

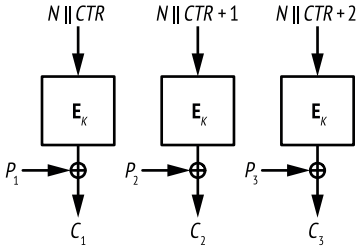

Рис. 4.10. Режим CTR

Листинг 4.6. Использование AES в режиме CTR

```
#!/usr/bin/env python

from Crypto.Cipher import AES
from Crypto.Util import Counter
from binascii import hexlify as hexa
from os import urandom
from struct import unpack

k = urandom(16)
print “k = %s” % hexa(k)

# выбрать одноразовое число
nonce = unpack(‘<Q’, urandom(8))[0]

# создать объект счетчика
ctr = Counter.new(128, initial_value=nonce)
# создать экземпляр AES в режиме CTR, используя ctr в качестве счетчика
aes = AES.new(k, AES.MODE_CTR, counter=ctr)

# в режиме CTR шифровать целый блок необязательно
p = ‘\x00\x01\x02\x03’

# зашифровать p
c = aes.encrypt(p)
print “enc(%s) = %s” % (hexa(p), hexa(c))

# дешифрировать, используя функцию шифрования
ctr = Counter.new(128, initial_value=nonce)
aes = AES.new(k, AES.MODE_CTR, counter=ctr)
p = aes.encrypt(c)
print “enc(%s) = %s” % (hexa(c), hexa(p))
```

В этом примере шифруется 4-байтовый открытый текст, и в результате получается 4-байтовый шифртекст. Затем шифртекст дешифрируется с применением функции шифрования:

```
$ ./aes_ctr.py
k = 130a1aa77fa58335272156421cb2a3ea
enc(00010203) = b23d284e
enc(b23d284e) = 00010203
```

Как и начальное значение в режиме CBC, одноразовое значение в режиме CTR задается шифрующей стороной и передается вместе с шифртекстом в открытом виде. Но, в отличие от начального значения CBC, от одноразового значения CTR не требуется случайности, оно просто должно быть уникальным. А уникальность необходима по той же причине, по которой нельзя дважды использовать одноразовый блокнот: если P_1 зашифровывается в $C_1 = P_1 \oplus S$ с помощью псевдослучайного потока S, а затем P_2 зашифровывается в $C_2 = P_2 \oplus S$ с помощью потока с тем же одноразовым числом, то, зная $C_1 \oplus C_2$, мы можем узнать $P_1 \oplus P_2$.

Случайное одноразовое число годится, если оно достаточно длинное: например, если его длина равна n бит, то велики шансы, что после $2^{n/2}$ шифрований и генерирования такого же количества одноразовых чисел встретятся дубликаты. Поэтому 64 бит недостаточно, т. к. можно ожидать повторения после приблизительно 2^{32} одноразовых чисел, а это недопустимо часто.

Счетчик гарантированно будет уникален, если он увеличивается на единицу для каждого нового открытого текста и при этом достаточно длинный, например 64-битовый.

Преимущество CTR в том, что он может работать быстрее любого другого режима. Мало того что он допускает распараллеливание, так еще начать шифрование можно, даже не получив сообщения, достаточно выбрать одноразовое число и вычислить поток, который впоследствии будет объединен с открытым текстом операцией XOR.

Какие возможны проблемы

Существует две атаки на блочные шифры, о которых необходимо знать: встреча посередине – техника, открытая в 1970-х годах, но все еще используемая во многих криптоаналитических атаках (не путать с атаками с человеком посередине), и класс атак на оракул дополнения, открытых в 2002 году криптографами из академических учреждений, затем позабытых и заново открытых через десять лет вместе с несколькими уязвимыми приложениями.

Атаки типа встречи посередине

Блочный шифр 3DES – это усовершенствованный вариант стандарта 1970-х годов DES, который принимает ключ длиной $56 \times 3 = 168$ бит (лучше, чем 56-битовый ключ в DES). Но уровень безопасности 3DES составляет 112, а не 168 бит из-за атаки «*встреча посередине*» (meet-in-the-middle – MitM).

Как показано на рис. 4.11, 3DES шифрует блок с использованием функций шифрования и дешифрирования DES: сначала шифрование с ключом K_1, затем дешифрирование с ключом K_2 и, наконец, еще одно шифрование с ключом K_3. Если $K_1 = K_2$, то первые два вызова взаимно уничтожаются и 3DES сводится к DES с ключом K_3. После-

довательность шифрование–дешифрирование–шифрование применяется вместо тройного шифрования, чтобы системы могли при необходимости эмулировать DES, пользуясь новым интерфейсом 3DES.

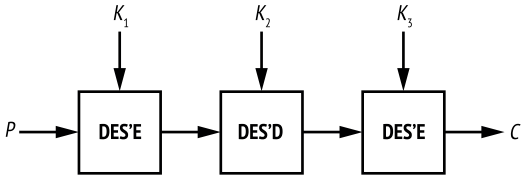

Рис. 4.11. Конструирование блочного шифра 3DES

Для чего нужно использовать тройной DES, а не просто два раза DES, т. е. $\mathbf{E}(K_2, \mathbf{E}(K_1, P))$? Как выясняется, из-за атаки MitM двойной DES небезопаснее простого. На рис. 4.12 показана атака MitM в действии.

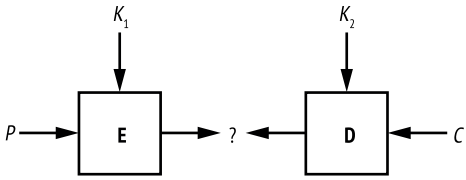

Рис. 4.12. Атака «встреча посередине»

Для вскрытия двойного DES атака «встреча посередине» применяется следующим образом:

1. Пусть даны P и $C = \mathbf{E}(K_2, \mathbf{E}(K_1, P))$ с двумя неизвестными 56-битовыми ключами K_1 и K_2. (DES принимает 56-битовые ключи, поэтому двойной DES принимает 112 бит ключей.) Построим таблицу ключ-значение, содержащую 2^{56} элементов вида $\mathbf{E}(K_1, P)$, где **E** – функция шифрования DES, а K_1 – хранимое значение.
2. Для всех 2^{56} возможных значений K_2 вычислим $\mathbf{D}(K_2, C)$ и проверим, является ли результат одним из индексов таблицы (среднее значение, представленное вопросительным знаком на рис. 4.12).
3. Если среднее значение является индексом таблицы, то мы находим соответствующее ему значение K_1 из таблицы и проверяем, что найденная пара ключей (K_1, K_2) правильная, на других парах P и C. Достаточно зашифровать P ключами K_1 и K_2 и убедиться, что полученный шифртекст совпадает с C.

Этот метод восстанавливает K_1 и K_2 путем выполнения 2^{57}, а не 2^{112} операций: на шаге 1 шифруется 2^{56} блоков, а затем на шаге 2 дешифрируется не более 2^{56} блоков, так что всего получается $2^{56} + 2^{56} = 2^{57}$ операций. Кроме того, нужно будет хранить 2^{56} элементов по 15 байт каждый, т. е. примерно 1 экзабайт. Это много, но существует трюк, позволяющий провести эту атаку, потребляя пренебрежимо мало памяти (см. главу 6).

Как видим, атаку MitM можно применить к 3DES почти так же, как и к двойному DES, только на третьем этапе нужно будет перебрать все 2^{112} комбинаций K_2 и K_3. Поэтому атака заведомо достигнет успеха после 2^{112} операций, а значит, 3DES обеспечивает только 112-битовую безопасность, хотя длина ключа равна 168 бит.

Атаки на оракул дополнения

В заключение этой главы рассмотрим одну из самых простых и вместе с тем самых разрушительных атак 2000-х годов: атаку на оракул дополнения. Напомним, что под дополнением понимается включение лишних байтов в последний блок открытого текста. Например, открытый текст, содержащий 111 байт, состоит из шести 16-байтовых блоков, за которыми следует еще 15 байт. Для формирования полного блока нужно добавить байт 01. В случае 110-байтового блока нужно добавить два байта 02 и т. д.

Оракул дополнения – это система, которая ведет себя по-разному в зависимости от того, является ли корректным дополнение в шифртексте, созданном в режиме CBC. Можно рассматривать его как черный ящик или API, возвращающий код *успеха* или *ошибки*. Оракул дополнения можно найти в любой службе на удаленном сервере, которая отправляет сообщения об ошибке, получив некорректно сформированный шифртекст. При заданном оракуле дополнения атака на него запоминает, какие входные данные дополнены правильно, а какие нет, и использует эту информацию, чтобы дешифрировать подобранные шифртексты.

Допустим, мы хотим дешифрировать блок C_2 шифртекста. Обозначим X искомое значение, т. е. $\mathbf{D}(K, C_2)$, и пусть P_2 – блок, полученный после дешифрирования в режиме CBC (см. рис. 4.13). Если выбрать случайный блок C_1 и отправить состоящий из двух блоков шифртекст $C_1 || C_2$ оракулу, то дешифрирование окажется успешным, только если $C_1 \oplus P_2 = X$ заканчивается корректным дополнением – одним байтом 01, двумя байтами 02, тремя байтами 03 и т. д.

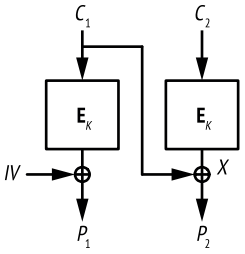

Рис. 4.13. Атака на оракул дополнения восстанавливает X путем выбора C_1 и проверки корректности дополнения

Основываясь на этом наблюдении, можно с помощью атаки на оракул дополнения дешифрировать блок C_2, зашифрованный в режиме CBC, следующим образом (байты обозначены, как в массиве: $C_1[0]$ – первый байт C_1, $C_1[1]$ – второй байт, ..., $C_1[15]$ – последний байт):

1. Выбрать случайный блок C_1 и изменять его последний байт, пока оракул дополнения не сочтет шифртекст корректным. Обычно в корректном шифртексте $C_1[15] \oplus X[15] = 01$, так что, опробовав 128 значений $C_1[15]$, мы найдем $X[15]$.
2. Определить значение $X[14]$, положив $C_1[15]$ равным $X[15] \oplus 02$ и найдя такой байт $C_1[14]$, который приводит к правильному дополнению. Как только оракул счел шифртекст корректным, мы нашли $C_1[14]$ такой, что $C_1[14] \oplus X[14] = 02$.
3. Повторить шаги 1 и 2 для всех 16 байт.

Для атаки нужно в среднем 128 запросов к оракулу на каждый из 16 байт, т. е. всего 2000 запросов. (Отметим, что во всех запросах следует использовать одно и то же начальное значение.)

Примечание *На практике реализация атаки на оракул дополнения несколько сложнее, потому что нужно обрабатывать неверные гипотезы на шаге 1. Шифртекст может иметь корректное дополнение не потому, что P_2 оканчивается одним байтом 01, а потому, что заканчивается двумя байтами 02 или тремя байтами 03. Но с этим легко справиться, проверяя корректность шифртекстов, в которых модифицируется больше байтов.*

Для дополнительного чтения

О блочных шифрах можно говорить еще очень много: и о том, как работают алгоритмы, и о том, как их можно атаковать. Например, схема Фейстеля и подстановочно-перестановочные сети – не единственные способы построения блочных шифров. В шифрах IDEA и FOX используется построение Лая-Месси, а в шифре Threefish – сети ARX, включающие комбинации сложения, циклического сдвига и XOR.

Существует также много других режимов, помимо ECB, CBC и CTR. Некоторые из них – чистая теория, их никто не использует, например CFB и OFB, другие предназначены для специализированных приложений, например XTS для настраиваемого шифрования или GCM для шифрования с аутентификацией.

Я обсудил шифр Rijndael, победивший в конкурсе на звание AES, но в нем принимали участие еще 14 алгоритмов: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, Magenta, MARS, RC6, SAFER+, Serpent и Twofish. Рекомендую поинтересоваться, как они работают, как проектировались, каким атакам подвергались и насколько они быстрые. Стоит также почитать о наработках АНБ (Skipjack и более поздних, SIMON и SPECK), а также о сравнительно недавних «облегченных» шифрах, например KATAN, PRESENT или PRINCE.

5

ПОТОКОВЫЕ ШИФРЫ

Симметричные шифры бывают блочными или потоковыми. Напомним (см. главу 4), что блочные шифры перемешивают блоки битов открытого текста с битами ключа и порождают блоки шифртекста такого же размера, обычно 64 или 128 бит. С другой стороны, потоковые шифры ничего не перемешивают, а генерируют псевдослучайные биты по ключу и шифруют открытый текст, применяя к нему и этим псевдослучайным битам операцию XOR, – принцип такой же, как в одноразовом блокноте, описанном в главе 1.

Потоковых шифров иногда чураются, потому что исторически они были не столь надежными, как блочные, и чаще взламывались. Это относится и к экспериментальным шифрам, спроектированным любителями, и к шифрам, применяемым в системах, которые развернуты в миллионах экземпляров, включая мобильные телефоны, Wi-Fi и смарт-карты для оплаты проезда в общественном транспорте. По счастью, хотя для этого понадобилось 20 лет, теперь мы знаем, как

проектировать безопасные потоковые шифры, и доверяем им защищать такие вещи, как подключения по Bluetooth, мобильную связь в системах 4G, TLS-соединения и многое другое.

В этой главе мы сначала опишем, как работают потоковые шифры, и обсудим два основных класса таких шифров: с хранимым состоянием и на основе счетчика. Затем мы изучим аппаратные и программные потоковые шифры и рассмотрим некоторые небезопасные шифры (например, A5/1 в стандарте мобильной связи GSM и RC4 в TLS) и некоторые современные безопасные шифры (например, Grain-128a для применения в оборудовании и Salsa20 для применения в программах).

Как работают потоковые шифры

Потоковые шифры ближе к детерминированным генераторам случайных битов (DRBG), чем к полноценным генераторам псевдослучайных чисел (PRNG), потому что, как и DRBG, они детерминированы. Детерминированность потоковых шифров позволяет осуществлять дешифрирование путем повторного генерирования тех же случайных битов, которые использовались при шифровании. PRNG позволил бы зашифровать сообщение, но не позволил бы дешифрировать его – это безопасно, но и бесполезно.

От DRBG потоковые шифры отличаются тем, что DRBG принимает одно входное значение, тогда как потоковые шифры – два: ключ и одноразовое число (nonce). Ключ должен быть секретным и обычно состоит из 128 или 256 бит. Одноразовое число может быть несекретным, но должно быть уникально для каждого ключа, обычно его длина варьируется от 64 до 128 бит.

Потоковые шифры порождают псевдослучайный поток битов, называемый *ключевым потоком*, или *гаммой*. Гамма объединяется с открытым текстом операцией XOR для шифрования, а затем та же самая гамма применяется к шифртексту для дешифрирования. На рис. 5.1 показана базовая операция шифрования в потоковом шифре, где **SC** – алгоритм шифра, *KS* – гамма, *P* – открытый текст, *C* – шифртекст.

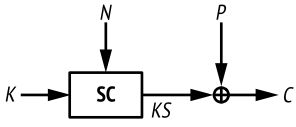

Рис. 5.1. Потоковый шифр производит шифрование, принимая ключ K и открытое одноразовое число N

Потоковый шифр вычисляет $KS = \mathbf{SC}(K, N)$, зашифровывает открытый текст операцией $C = P \oplus KS$ и дешифрирует шифртекст операцией $P = C \oplus KS$. Функции шифрования и дешифрирования одинаковы, потому что делают одно и то же: вычисляют XOR битов и гаммы. Именно поэтому в некоторых криптографических библиотеках имеется единственная функция `encrypt`, которая применяется как для шифрования, так и для дешифрирования.

Потоковый шифр позволяет зашифровать одно сообщение ключом K_1 и одноразовым числом N_1, а затем другое сообщение – ключом K_1 и одноразовым числом N_2, отличным от N_1, или ключом K_2, отличным от K_1, и одноразовым числом N_1. Однако никогда не следует повторно шифровать комбинацией K_1 и N_1, потому что в этом случае дважды использовалась бы одна и та же гамма *KS*. Тогда мы получили бы в свое распоряжение первый шифртекст $C_1 = P_1 \oplus KS$ и второй шифртекст $C_2 = P_2 \oplus KS$, и если P_1 известен, то можно было бы определить $P_2 = C_1 \oplus C_2 \oplus P_1$.

Примечание *Название «nonce» – это сокращение от «number used only once» (число, используемое только один раз). В контексте потоковых шифров его иногда называют IV, от «initial value» (начальное значение).*

Потоковые шифры с хранимым состоянием и на основе счетчика

На верхнем уровне есть два типа потоковых шифров: с хранимым состоянием и на основе счетчика. *Шифры с хранимым состоянием* имеют секретное внутреннее состояние, которое изменяется по мере генерирования гаммы. Состояние шифра инициализируется на основе ключа и одноразового числа, после чего вызывается функция обновления, которая изменяет состояние и порождает один или несколько битов гаммы, как показано на рис. 5.2. Например, знаменитый шифр RC4 является шифром с хранимым состоянием.

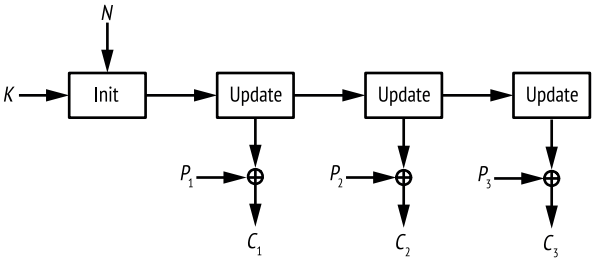

Рис. 5.2. Потоковый шифр с хранимым состоянием

Потоковые шифры на основе счетчика порождают порции гаммы из ключа, одноразового числа и значения счетчика, как показано на рис. 5.3. В отличие от шифров с хранимым состоянием, например Salsa20, в процессе генерирования гаммы никакое секретное состояние не запоминается.

Эти два подхода определяют верхнеуровневую архитектуру потокового шифра, не зависящую от способа работы базового алгоритма. По внутреннему устройству потоковые шифры также можно отнести к двум категориям в зависимости от целевой платформы: аппаратные и программные.

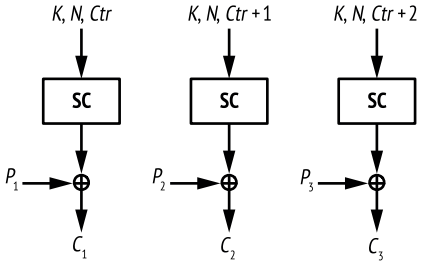

Рис. 5.3. Потоковый шифр на основе счетчика

Аппаратные потоковые шифры

Говоря об аппаратуре, криптографы имеют в виду специализированные заказные интегральные схемы (application-specific integrated circuit – ASIC), программируемые логические устройства (programmable logic device – PLD) и программируемые пользователем вентильные матрицы (field programmable gate array – FPGA, *рус.* ППВМ). Аппаратная реализация шифра – это электронная схема, реализующая криптографический алгоритм на битовом уровне, которая ни для чего другого не предназначена, т. е. это *специализированное оборудование*. С другой стороны, программные реализации криптографических алгоритмов просто говорят микропроцессору, какие команды нужно выполнить, чтобы алгоритм работал. Эти команды применяются к байтам или словам и активируют те части электронной схемы, которые отвечают за операции общего назначения, например сложение и умножение. Программа имеет дело с байтами или словами длиной 32 или 64 бита, а оборудование – с битами. Первые потоковые шифры работали с битами, чтобы уйти от сложных операций со словами и увеличить эффективность при аппаратной реализации, на которую в то время и были рассчитаны.

Главная причина, по которой аппаратные реализации потоковых шифров были так широко распространены, – дешевизна по сравнению с блочными шифрами. Для потоковых шифров нужно меньше памяти и меньше логических элементов, поэтому они занимают меньше места на интегральной схеме, а значит, сокращается стоимость изготовления. Например, в терминах эквивалентных логических элементов, стандартной меры площади для интегральных схем, потоковый шифр занимает меньше 1000 эквивалентных логических элементов, тогда как типичный программно-ориентированный блочный шифр требует по меньшей мере 10 000 эквивалентных логических элементов, т. е. шифрование обходится на порядок дороже.

Однако в наше время блочные шифры уже стоят не дороже потоковых; во-первых, потому что появились пригодные для аппаратной реализации блочные шифры, требующие не больше площади, чем потоковые, а во-вторых, потому что стоимость оборудования резко упала.

В следующем разделе я объясню основной механизм, лежащий в основе аппаратных потоковых шифров, – регистры сдвига с обрат-

ной связью (feedback shift register – FSR). Почти все аппаратные потоковые шифры так или иначе опираются на FSR, будь то шифр A5/1, который использовался в мобильных телефонах стандарта 2G, или более современный шифр Grain-128a.

Примечание *Первый стандартный блочный шифр, Data Encryption Standard (DES), был оптимизирован для аппаратной, а не программной реализации. Когда в 1970-х годах правительство США стандартизовало DES, большая часть приложений, в которых он использовался, была реализована аппаратно. Поэтому неудивительно, что S-блоки в DES небольшие и быстрые, если реализовывать их в виде электронной схемы, но неэффективны в случае программной реализации. В отличие от DES, текущий стандарт Advanced Encryption Standard (AES) работает с байтами и потому программно реализуется более эффективно, чем DES.*

Регистры сдвига с обратной связью

В бесчисленных потоковых шифрах использовались FSR, потому что они просты и понятны. FSR – это просто массив битов, снабженный *функцией обратной связи*, которую я буду обозначать **f**. Состояние FSR хранится в массиве, или регистре, а при каждом *обновлении* FSR вызывается функция обратной связи, которая изменяет состояние и порождает один выходной бит.

На практике FSR работает следующим образом: если R_0 – начальное значение FSR, то следующее состояние R_1 определяется как результат сдвига R_0 влево на 1 бит. При этом бит, выдвигающийся из регистра, возвращается в виде выходного, а в освободившуюся позицию помещается $\mathbf{f}(R_0)$.

То же правило применяется для вычисления следующих состояний R_2, R_3 и т. д. То есть если известно состояние FSR в момент t, R_t, то следующее состояние R_{t+1} вычисляется по формуле

$$R_{t+1} = (R_t << 1) \mid \mathbf{f}(R_t).$$

В этой формуле | обозначает оператор логического ИЛИ (OR), а << – оператор сдвига влево (как в языке C). Например, для 8-битовой строки 00001111 имеем:

```
00001111 << 1 = 00011110
00011110 << 1 = 00111100
00111100 << 1 = 01111000
```

Операция сдвига сдвигает биты влево, при этом самый левый бит теряется, а самый правый обнуляется. Операция обновления в FSR устроена аналогично, только в правый бит записывается не 0, а значение $\mathbf{f}(R_t)$.

Рассмотрим, к примеру, 4-битовый FSR, для которого функция обратной связи **f** заключается в применении XOR ко всем четырем битам. Инициализируем состояние следующим образом:

1 1 0 0

Теперь сдвинем биты влево, тогда на выходе получится 1, а самый правый бит примет значение

$$\mathbf{f}(1100) = 1 \oplus 1 \oplus 0 \oplus 0 = 0.$$

После этого состояние становится равным

1 0 0 0

Следующее обновление выводит 1, сдвигает состояние влево и записывает в правый бит значение

$$\mathbf{f}(1000) = 1 \oplus 0 \oplus 0 \oplus 0 = 1.$$

Теперь состояние равно

0 0 0 1

Следующие три обновления выводят три нуля и приводят к такой череде состояний:

0 0 1 1
0 1 1 0
1 1 0 0

Таким образом, после пяти итераций мы возвращаемся к начальному состоянию 1100, и легко понять, что цикл, начинающийся в любом из наблюдавшихся состояний, через пять итераций вернется к нему же. Говорят, что 5 является *периодом* FSR для любого из начальных состояний 1100, 1000, 0001, 0011, 0110. Поскольку период этого FSR равен 5, за 10 тактов регистр дважды породит одну и ту же 5-битовую последовательность. А из начального состояния 1100 за 20 тактов регистр выведет биты 11000110001100011000, т. е. четыре раза повторит одну и ту же 5-битовую последовательность 11000. Интуитивно представляется, что таких повторяющихся последовательностей следует избегать, и чем больше период, тем безопаснее.

Примечание *Если вы планируете использовать FSR в потоковом шифре, то избегайте регистров с короткими периодами, поскольку они делают вывод более предсказуемым. Для одних типов FSR вычислить период легко, а для других почти невозможно.*

На рис. 5.4 показана структура этого цикла, а также другие циклы данного FSR; каждый цикл представлен окружностью, на которой состояния регистра отмечены точками.

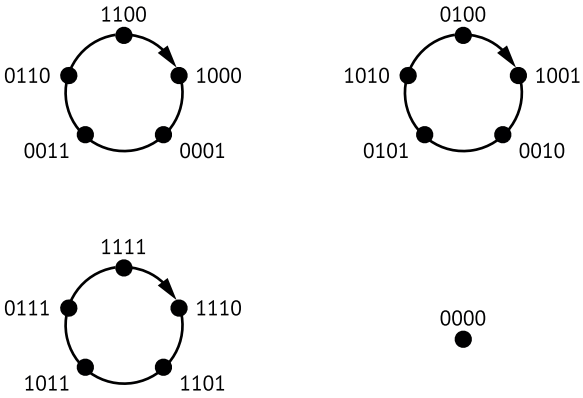

Рис. 5.4. Циклы FSR, функция обратной связи которого объединяет все 4 бита операцией XOR

В действительности этот конкретный FSR имеет два других цикла с периодом 5, а именно {0100, 1001, 0010, 0101, 1010} и {1111, 1110, 1101, 1011, 0111}. Заметим, что любое состояние принадлежит только одному циклу. В данном случае налицо три цикла по пять состояний в каждом, покрывающие 15 из $2^4 = 16$ возможных значений нашего 4-битового регистра. Шестнадцатое значение равно 0000, на рис. 5.4 оно показано циклом периода 1, потому что FSR переводит 0000 в 0000.

Мы видели, что FSR по существу является битовым регистром, при каждом обновлении которого выводится один бит (самый левый), а функция вычисляет новое значение самого правого бита (все остальные биты сдвигаются влево). Период FSR в некотором начальном состоянии – это количество обновлений, необходимое для перехода FSR в то же самое состояние. Если для этого требуется N обновлений, значит, FSR будет порождать одну и ту же последовательность N бит снова и снова.

Регистры сдвига с линейной обратной связью

Регистры сдвига с линейной обратной связью (linear feedback shift register – LFSR) – это FSR с *линейной* функцией обратной связи, т. е. функцией, которая применяет операцию XOR к некоторым битам состояния, как в примере из предыдущего раздела, где функция обратной связи возвращала результат применения XOR ко всем четырем битам регистра. Напомним, что в криптографии линейность – синоним предсказуемости и говорит о простой математической структуре. И естественно, что LFSR допускают анализ с помощью таких понятий,

как линейная сложность, конечные поля и примитивные полиномы. Но я опущу детали, а приведу лишь основные факты.

Решение о том, к каким битам применять XOR, определяет период LFSR и потому критически важно для его криптографической ценности. Хорошая новость заключается в том, что мы знаем, как выбирать позиции битов, гарантирующие максимальный период $2^n - 1$. А именно нужно взять индексы битов, от 1 для самого правого до n для самого левого, и выписать полиномиальное выражение $1 + X + X^2 + \ldots + X^n$, где член X^i включается, только если i-й бит входит в число тех, к которым применяется XOR в функции обратной связи. Период будет максимальным *тогда и только тогда, когда* этот полином *примитивен*. Полином называется примитивным, если обладает следующими свойствами:

- полином должен быть неприводимым, т. е. его нельзя разложить на множители – записать в виде произведения полиномов меньшей степени. Например, $X + X^3$ не является неприводимым, потому что раскладывается в произведение $(1 + X)(X + X^2)$:

$$(1 + X)(X + X^2) = X + X^2 + X^2 + X^3 = X + X^3;$$

- полином должен обладать некоторыми математическими свойствами, которые трудно объяснить, не вводя нетривиальных понятий, но легко проверить.

Примечание *Максимальный период n-битового LFSR равен $2^n - 1$, а не 2^n, потому что состояние, в котором все биты равны нулю, повторяется бесконечно. Поскольку применение XOR к любому количеству нулей дает нуль, новые биты, вдвигающиеся в состояние справа, всегда будут равны нулю, так что из нулевого состояния FSR непременно переходит в него же.*

Например, на рис. 5.5 показан 4-битовый LFSR с полиномиальной обратной связью $1 + X + X^3 + X^4$, которая применяет XOR к битам в позициях 1, 3 и 4 для вычисления нового значения бита L_1. Однако этот полином не является примитивным, потому что разлагается в произведение $(1 + X^3)(1 + X)$.

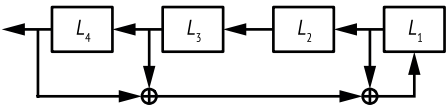

Рис. 5.5. LFSR с полиномиальной обратной связью $1 + X + X^3 + X^4$

И действительно, период LFSR, показанного на рис. 5.5, не максимален. Чтобы доказать это, начнем с состояния 0001.

0 0 0 1

Теперь сдвинем на 1 бит влево и присвоим правому биту значение 0 + 0 + 1 = 1:

0 0 1 1

Повторим эту операцию четыре раза, получим такую последовательность состояний:

0 1 1 1
1 1 1 0
1 1 0 0
1 0 0 0

Как видим, после шести обновлений мы вернулись в начальное состояние, т. е. этот LFSR имеет период 6, а не максимальный период 15.

Для контраста рассмотрим LFSR на рис. 5.6.

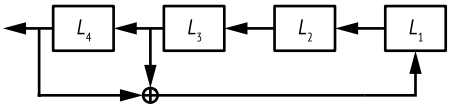

Рис. 5.6. LFSR с функцией обратной связи $1 + X^3 + X^4$, примитивным полиномом, гарантирующим максимальный период

Эта обратная связь описывается примитивным полиномом $1 + X^3 + X^4$, и легко проверить, что период LFSR максимален (равен 15). В самом деле, имеем такую последовательность состояний:

0 0 0 1	0 0 1 1	0 1 0 1	1 1 1 0
0 0 1 0	0 1 1 0	1 0 1 1	1 1 0 0
0 1 0 0	1 1 0 1	0 1 1 1	1 0 0 0
1 0 0 1	1 0 1 0	1 1 1 1	0 0 0 1

В ней встречаются все возможные состояния, кроме 0000, и ни одно состояние не повторяется. Это доказывает, что период максимален, а полином обратной связи примитивен.

К сожалению, использование LFSR в качестве потокового шифра небезопасно. Если n – длина LFSR в битах, то противнику нужно знать всего n выходных бит, чтобы восстановить начальное состояние LFSR, а следовательно, определить все предыдущие биты и предсказать все будущие. Эта атака возможна, потому что алгоритм Берлекэмпа-Месси позволяет решить уравнения, описывающие математическую структуру LFSR, и найти не только его начальное состояние, но и полиномиальную функцию обратной связи. На самом деле даже не нужно знать точную длину LFSR; можно повторять алгоритм Берлекэмпа-Месси для всех возможных значений n, пока не будет найдено правильное.

Подводя итоги, констатируем, что LFSR криптографически нестойкие в силу своей линейности. Выходные биты и биты начального состояния связаны простыми и короткими уравнениями, которые легко решаются школьными линейно-алгебраическими методами. Для повышения стойкости LFSR необходимо добавить нелинейность.

Фильтрующий LFSR

Чтобы справиться с небезопасностью LFSR, мы можем скрыть их линейность, пропустив выходные биты через нелинейную функцию перед возвратом, – получится *фильтрующий LFSR* (см. рис. 5.7).

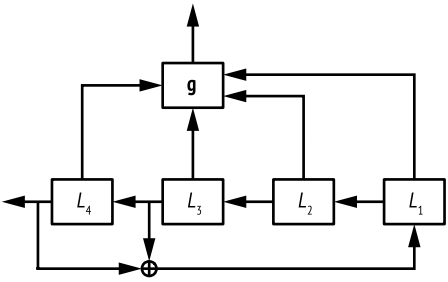

Рис. 5.7. Фильтрующий LFSR

Функция **g** на рис. 5.7 должна быть *нелинейной* – она объединяет биты с помощью XOR и комбинирует их с помощью логических операций AND и OR. Например, $L_1L_2 + L_3L_4$ – нелинейная функция (я опустил знак умножения, т. е. L_1L_2 означает $L_1 \times L_2$, или L_1 & L_2 в нотации C).

Примечание *Отметим, что функции обратной связи можно записывать непосредственно в терминах битов FSR, как в $L_1L_2 + L_3L_4$, или применяя эквивалентную полиномиальную нотацию $1 + XX^2 + X^3X^4$. Прямая нотация нагляднее, а полиномиальная удобнее для математического анализа свойств FSR. Мы будем придерживаться прямой нотации, пока не понадобится коснуться математических свойств.*

Фильтрующие LFSR криптографически более стойкие, чем обычные, потому что нелинейная функция препятствует прямолинейным атакам. Тем не менее более сложные атаки типа описанных ниже способны вскрыть систему.

- В случае *алгебраических атак* решается система нелинейных уравнений, построенная по выходным битам. Неизвестными в этих уравнениях являются биты состояния LFSR.
- В случае *кубических атак* вычисляются производные нелинейных уравнений, чтобы уменьшить порядок системы до единицы, а затем эффективно решить ее как линейную.
- *Быстрые корреляционные атаки* направлены на функции фильтрации, которые, несмотря на нелинейность, во многих отношениях ведут себя как линейные функции.

Из этих примеров следует вынести простой урок – пластырем пулевые раны не залечишь. Латание непригодного алгоритма путем добавления чуть более стойкого уровня не сделает систему безопасной. Нужно устранить корень проблемы.

Нелинейные FSR

Нелинейные FSR (NFSR) похожи на LFSR, но функция обратной связи в них нелинейна. То есть в дополнение к поразрядным операциям XOR она может включать поразрядные AND и OR – свойство, у которого есть как плюсы, так и минусы.

Достоинства нелинейных функций обратной связи в том, что благодаря им NFSR оказываются криптографически более стойкими, чем LFSR, потому что выходные биты зависят от секретного начального состояния сложным образом и описываются уравнениями экспоненциального размера. В линейной функции связи просты и содержат не более n членов (N_1, N_2, ..., N_n, где N_i – биты состояния). Но, например, 4-битовый NFSR с секретным начальным состоянием (N_1, N_2, N_3, N_4) и функцией обратной связи ($N + N_2 + N_1N_2 + N_3N_4$) порождает первый выходной бит, равный

$$N_1 + N_2 + N_1N_2 + N_3N_4.$$

На второй итерации бит N_1 заменяется этим новым значением. Выражение второго выходного бита в терминах начального состояния имеет вид:

$$\begin{aligned}&(N_1N_2 + N_3N_4 + N_1 + N_2) + N_1 + (N_1N_2 + N_3N_4 + N_1 + N_2)N_1 + N_2N_3\\ &= N_1N_3N_4 + N_1N_2 + N_2N_3 + N_3N_4 + N_1 + N_2.\end{aligned}$$

Алгебраическая степень этого уравнения (максимальное число сомножителей в произведении, в данном случае в члене $N_1N_3N_4$) равна 3, тогда как степень функции обратной связи равна всего 2, а членов в уравнении шесть, а не четыре. Многократное применение нелинейной функции очень быстро порождает нерешаемые уравнения, поскольку размер выхода возрастает экспоненциально. Нам в процессе работы NFSR вычислять эти уравнения не нужно, но противнику придется решить их, если он хочет взломать систему.

Недостаток NFSR – отсутствие эффективного способа определить период регистра или хотя бы узнать, является ли его период максимальным. Для n-битового NFSR потребовалось бы провести примерно 2^n испытаний, чтобы проверить, максимален ли период. Для больших NFSR длиной 80 и более бит выполнить такое вычисление невозможно.

По счастью, есть способ использовать NFSR, не тревожась по поводу короткого периода: можно объединить LFSR и NFSR, получив в результате гарантированно максимальный период и криптографическую стойкость, – именно так работает шифр Grain-128a.

Grain-128a

Помните конкурс на звание AES, который мы обсуждали в главе 4 применительно к блочному шифру AES? Потоковый шифр Grain – продукт аналогичного конкурса под названием eSTREAM. Этот конкурс закрылся в 2008 году, и в результате был сформирован короткий список кандидатов на звание рекомендованного потокового шифра: четыре аппаратных и четыре программных. Grain – один из четырех аппаратных шифров, а Grain-128a – его усовершенствованная версия, разработанная теми же авторами. На рис. 5.8 показан механизм работы Grain-128a.

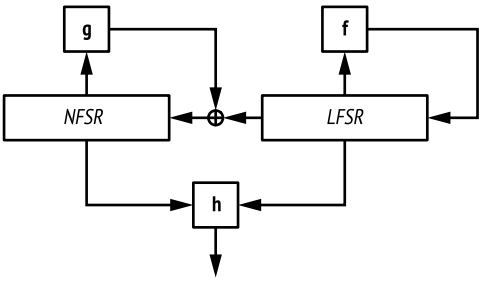

Рис. 5.8. Механизм работы шифра Grain-128a, состоящего из 128-битового NFSR и 128-битового LFSR

Как видно по рис. 5.8, шифр Grain-128a настолько прост, насколько это возможно для потокового шифра; он сочетает 128-битовый LFSR, 128-битовый NFSR и функцию фильтра **h**. LFSR имеет максимальный период $2^{128} - 1$, это гарантирует, что период всей системы не меньше $2^{128} - 1$, и, стало быть, защищает от потенциально коротких циклов в NFSR. В то же время NFSR и нелинейная функция фильтра **h** обеспечивают криптографическую стойкость.

Grain-128a принимает 128-битовый ключ и 96-битовое одноразовое число. Он копирует 128 бит ключа в 128 бит состояния NFSR, а 96 бит одноразового числа в первые 96 бит состояния LFSR, заполняя следующий 31 бит единицами и добавляя в конце один нуль. На этапе инициализации вся система обновляется 256 раз, и только после этого возвращается первый бит гаммы. Поэтому на этапе инициализации бит, возвращенный функцией **h**, не выводится в составе гаммы, а поступает в LFSR, гарантируя, что следующее состояние зависит одновременно от ключа и одноразового числа.

Функция обратной связи LFSR в шифре Grain-128a имеет вид

$$\mathbf{f}(L) = L_{32} + L_{47} + L_{58} + L_{90} + L_{121} + L_{128},$$

где $L_1, L_2, \ldots, L_{128}$ – биты LFSR. Эта функция использует только 6 бит из 128-битового LFSR, но этого достаточно для получения примитивного полинома, гарантирующего максимальный период. Благодаря небольшому числу битов уменьшается стоимость аппаратной реализации.

А вот как выглядит полином, описывающий обратную связь NFSR в шифре Grain-128a ($N1, \ldots, N128$):

$$\mathbf{g}(N) = N_{32} + N_{37} + N_{72} + N_{102} + N_{128} + N_{44}N_{60} + N_{61}N_{125} + N_{63}N_{67} + N_{69}N_{101} + N_{80}N_{80} + N_{110}N_{111} + N_{115}N_{117} + N_{46}N_{50}N_{58} + N_{103}N_{104}N_{106} + N_{33}N_{35}N_{36}N_{40}.$$

Эта функция тщательно подобрана, чтобы максимизировать криптографическую стойкость, одновременно минимизировав стоимость реализации. Ее алгебраическая степень равна 4 благодаря члену $N_{33}N_{35}N_{36}N_{40}$. Более того, **g** невозможно аппроксимировать линейной функцией, поскольку она сильно нелинейна. Наконец, в Grain-128a выходной бит LFSR подается на вход XOR вместе с выходом **g**, и результат становится новым значением правого бита NFSR.

Функция фильтра **h** также нелинейна, она принимает 9 бит от NFSR и 7 бит от LFSR и некоторым способом комбинирует их, добиваясь хороших криптографических свойств.

На момент написания данной книги неизвестно ни одной атаки на Grain-128a, и я уверен, что он так и останется безопасным. Grain-128a применяется во встраиваемых системах низкой ценовой категории, которые нуждаются в компактном и быстром потоковом шифре – в основном в промышленных коммерческих системах. Именно поэтому Grain-128a плохо известен сообществу ПО с открытым исходным кодом.

A5/1

A5/1 – потоковый шифр, который использовался для шифрования речи в стандарте мобильной связи 2G. Стандарт A5/1 был утвержден в 1987 году, но опубликован только в конце 1990-х годов после успешной обратной разработки. Атаки начались в начале 2000-х годов, и в конечном итоге A5/1 был взломан, да так, что можно было практически (а не только теоретически) дешифрировать зашифрованные разговоры. Посмотрим, почему и как такое могло случиться.

Механизм работы A5/1

A5/1 включает три LFSR, и в нем используется прием, который на первый взгляд выглядит очень изобретательно, но, как выясняется, не обеспечивает безопасность (см. рис. 5.9).

Как видим, в A5/1 используются LFSR с 19, 22 и 23 битами и следующими полиномами:

$$1 + X^{14} + X^{17} + X^{18} + X^{19};$$
$$1 + X^{21} + X^{22};$$
$$1 + X^{8} + X^{21} + X^{22} + X^{23}.$$

Как может быть безопасной схема, содержащая только LFSR и ни одного NFSR? Хитрость заключается в механизме обновления A5/1. Вместо того чтобы обновлять все три LFSR на каждом такте, разработчики A5/1 добавили следующее правило синхронизации.

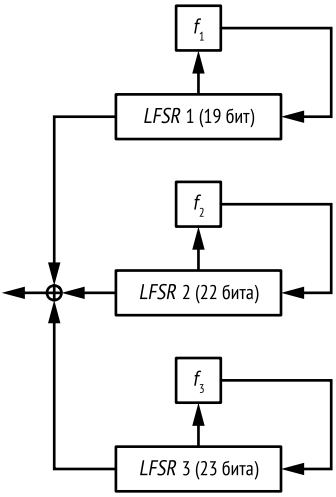

Рис. 5.9. Шифр A5/1

1. Проверить значение 9-го бита LFSR 1, 11-го бита LFSR 2 и 11-го бита LFSR 3, которые называются *битами синхронизации*. Либо все три бита имеют одинаковое значение (1 или 0), либо только два из них.
2. Сдвигаются лишь регистры, для которых биты синхронизации имеют такое же значение, как у большинства. При каждом обновлении сдвигаются два или три LFSR.

Без этого простого правила A5/1 не обеспечивал бы вообще никакой безопасности, а чтобы взломать шифр, нужно обойти это правило. Однако, как мы увидим, это проще сказать, чем сделать.

Примечание *В правиле нерегулярной синхронизации A5/1 каждый регистр сдвигается с вероятностью 3/4 при каждом обновлении. Действительно, вероятность, что хотя бы еще в одном регистре значение бита такое же, равна 1 – $(1/2)^2$, где $(1/2)^2$ – вероятность, что оба других бита синхронизации имеют другое значение.*

В стандарте 2G шифр A5/1 используется с 64-битовым ключом и 22-битовым одноразовым числом, которое изменяется для каждого нового кадра данных. Атаки на A5/1 вскрывают 64-битовое начальное состояние системы (19 + 22 + 23 бит начальных значений LFSR), что, в свою очередь, позволяет узнать одноразовое число (если оно еще не было известно) и ключ, раскрутив в обратную сторону механизм инициализации. Такие атаки называются *атаками с известным открытым текстом* (known-plaintext attack – KPA), потому что часть зашифрованных данных известна, что позволяет противнику определить соответствующие части гаммы, выполнив XOR шифртекста с известными кусками открытого текста.

Существует два основных типа атак на A5/1.

Тонкие атаки. Воспользоваться внутренней линейностью A5/1 и его простой системой нерегулярной синхронизации.

Грубые атаки. Воспользоваться только коротким ключом A5/1 и обратимостью внедрения номера кадра.

Посмотрим, как эти атаки работают.

Тонкие атаки

В тонкой атаке, получившей название *угадай-и-определи*, противник угадывает некоторые секретные части состояния, чтобы определить другие. В криптоанализе под «угадыванием» понимается полный перебор: для каждого возможного значения LFSR 1 и 2 и всех возможных значений бита синхронизации LFSR 3 на первых 11 тактах атака реконструирует биты LFSR 3 путем решения уравнений, зависящих от угаданных битов. Если догадка правильна, то противник получает правильное значение LFSR 3.

Псевдокод атаки выглядит следующим образом:

```
Для всех 2^19 значений начального состояния LFSR 1
    Для всех 2^22 значений начального состояния LFSR 2
        Для всех 2^11 бит синхронизации LFSR 3 на первых 11 тактах
            Реконструировать начальное состояние LFSR 3
            Проверить, верна ли догадка; если да, вернуться, иначе продолжить
```

Насколько эффективна эта атака по сравнению с полным перебором 2^{64} вариантов, который обсуждался в главе 3? В этой атаке число операций равно $2^{19} \times 2^{22} \times 2^{11} = 2^{52}$ в худшем случае, когда успешной оказывается самая последняя проверка. Это в 2^{12} (приблизительно в 4000) раз быстрее полного перебора в предположении, что последние две операции в приведенном выше псевдокоде требуют примерно столько же вычислений, сколько проверка 64-разрядного ключа при полном переборе. Но так ли это?

Вспомним обсуждение полной стоимости атаки в главе 3. При вычислении стоимости атаки нужно принимать во внимание не только объем вычислений, необходимых для ее проведения, но также возможность распараллеливания и потребление памяти. В данном случае ни с тем, ни с другим проблем не возникает: как и любая атака полным перебором, атака угадай-и-определи естественно параллельна (т. е. работает в N раз быстрее при выполнении на N ядрах) и требует не больше памяти, чем выполнение самого шифра.

Наша оценка стоимости 2^{52} неточна по другой причине. На самом деле каждая из 2^{52} операций (проверка потенциального ключа) занимает примерно в четыре раза больше тактов, чем проверка ключа в атаке полным перебором. В итоге получается, что реальная стоимость этой конкретной атаки ближе к $4 \times 2^{52} = 2^{54}$ операциям.

С помощью атаки угадай-и-определи на шифр A5/1 можно дешифрировать мобильную связь, но для определения ключа понадобится два часа работы в кластере специализированного оборудования. Иными словами, ни о каком дешифрировании в режиме реального

времени и речи быть не может. Для этого нам понадобится атака другого типа.

Грубые атаки

Атака с компромиссом между временем и памятью (TMTO) является грубой атакой на A5/1. Ей безразлично внутреннее устройство A5/1; важно только, что размер его состояния 64 бита. Атака TMTO рассматривает A5/1 как черный ящик, который принимает 64-битовое значение (состояние) и выдает 64-битовое значение (первые 64 бита гаммы).

Идея атаки заключается в том, чтобы уменьшить стоимость полного перебора ценой использования очень большого объема памяти. Простейший тип TMTO – атака по кодовой книге. В этом случае мы заранее вычисляем таблицу, содержащую 2^{64} элементов – комбинации ключа и значения (*key:value*), и сохраняем выходное значение для каждого из 2^{64} возможных ключей. Чтобы воспользоваться этой предварительно вычисленной таблицей для атаки, мы просто берем выход экземпляра A5/1 и по таблице смотрим, какой ключ соответствует этому выходу. Сама атака работает очень быстро – занимает столько времени, сколько нужно для поиска значения в памяти, – но для создания таблицы требует выполнить 2^{64} вычислений A5/1. Хуже того, атаки по кодовой книге потребляют безумный объем памяти: $2^{64} \times (64 + 64)$ бит, т. е. 2^{68} байт, или 256 экзабайт. Это емкость десятков центров обработки данных, так что не стоит даже глядеть в эту сторону.

TMTO-атаки уменьшают объем памяти, необходимой для атаки по кодовой книге, за счет увеличенного количества вычислений в онлайновой фазе атаки; чем меньше таблица, тем больше вычислений нужно для вскрытия ключа. Для подготовки таблицы все равно придется выполнить 2^{64} операций, но это делается только один раз.

В 2010 году исследователи потратили около двух месяцев на построение таблиц объемом два терабайта с помощью графических процессоров (GPU) и параллельного выполнения 100 000 экземпляров A5/1. При наличии таких больших таблиц вызовы, зашифрованные A5/1, удалось дешифрировать почти в реальном масштабе времени. Сотовые операторы реализовали временные заплаты для противодействия этой атаке, но настоящее решение пришло только с принятием стандартов мобильной телефонии 3G и 4G, из которых A5/1 вообще был устранен.

Программные потоковые шифры

Программные потоковые шифры работают с 32- или 64-битовыми словами, а не с отдельными битами, поскольку это более эффективно на современных процессорах, где команды могут выполнять арифметические операции над словом так же быстро, как над одним битом. Поэтому программные потоковые шифры лучше приспособлены для

исполнения серверами или браузерами, работающими на персональных компьютерах, где мощные процессоры общего назначения выполняют алгоритм шифра как обычную программу.

В настоящее время значительный интерес к программным потоковым шифрам объясняется несколькими причинами. Во-первых, поскольку многие устройства оснащаются мощными процессорами и оборудование дешевеет, отпадает нужда в компактных бит-ориентированных шифрах. Например, оба потоковых шифра, являющихся частью стандарта мобильной связи 4G (европейский SNOW3G и китайский ZUC), оперируют 32-битовыми словами, а не битами, как устаревший A5/1.

Во-вторых, программные потоковые шифры приобрели популярность в противовес блочным шифрам, и особенно после фиаско с атакой на оракул дополнения против блочных шифров в режиме CBC. Кроме того, потоковые шифры проще специфицировать и реализовать, чем блочные: вместо смешивания битов сообщения и ключа потоковые шифры просто вставляют биты ключа в качестве секрета. На самом деле один из самых популярных потоковых шифров фактически является замаскированным блочным шифром: AES в режиме счетчика (CTR).

В одном из способов построения потоковых шифров, применяемом в SNOW3G и ZUC, копируются аппаратные шифры с их FSR, только биты заменяются байтами или словами. Но не они представляют наибольший интерес для криптографа. На момент написания этой книги наибольший интерес вызывали шифры RC4 и Salsa20, используемые в многочисленных системах, несмотря на то что один из них полностью взломан.

RC4

Спроектированный в 1987 году Роном Ривестом из компании RSA Security, затем подвергнутый обратной разработке и опубликованный во всех деталях в 1994 году, RC4 долгое время был самым популярным потоковым шифром. RC4 использовался в бесчисленных приложениях, из которых самые известные – первый стандарт шифрования Wi-Fi, Wired Equivalent Privacy (WEP), и протокол Transport Layer Security (TLS), применяемый для установления HTTPS-соединений. К сожалению, RC4 недостаточно безопасен для большинства приложений, включая WEP и TLS. Чтобы понять, почему, посмотрим, как RC4 работает.

Как работает RC4

RC4 – один из самых простых когда-либо созданных шифров. В нем нет никаких криптографических операций – ни XOR, ни умножений, ни S-блоков… ничего. Он просто меняет местами байты. Внутренним состоянием RC4 является массив S из 256 байт, элементы которого первоначально равны $S[0] = 0$, $S[1] = 1$, $S[2] = 2$, …, $S[255] = 255$, а за-

тем инициализируются на основе n-байтового ключа K с помощью *алгоритма развертки ключа* (key scheduling algorithm – KSA), работа которого иллюстрируется Python-кодом в листинге 5.1.

Листинг 5.1. Алгоритм развертки ключа в RC4

```
j = 0
# инициализировать массив S: S[0] = 0, S[1] = 1, … , S[255] = 255
S = range(256)
# цикл по i от 0 до 255
for i in range(256):
    # вычислить сумму v
    j = (j + S[i] + K[i % n]) % 256
    # обменять S[i] и S[j]
    S[i], S[j] = S[j], S[i]
```

После завершения этого алгоритма массив S по-прежнему содержит все значения от 0 до 255, но уже в порядке, который выглядит как случайный. Например, если 128-битовый ключ состоит из одних нулей, то состояние S (элементы от $S[0]$ до $S[255]$) выглядит так:

0, 35, 3, 43, 9, 11, 65, 229, (…), 233, 169, 117, 184, 31, 39.

Но если инвертировать первый бит ключа и снова выполнить KSA, то получится совершенно другое состояние, тоже выглядящее случайным:

32, 116, 131, 134, 138, 143, 149, (…), 152, 235, 111, 48, 80, 12.

Имея начальное состояние S, RC4 генерирует гамму KS такой же длины, как открытый текст P, и с ее помощью вычисляет шифртекст: $C = P \oplus KS$. Байты гаммы KS вычисляются по S, как показано в листинге 5.2, где предполагается, что длина P равна m байт.

Листинг 5.2. Генерирование гаммы в RC4, где S – начальное состояние, инициализированное в листинге 5.1

```
i = 0
j = 0
for b in range(m):
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    S[i], S[j] = S[j], S[i]
    KS[b] = S[(S[i] + S[j]) % 256]
```

В листинге 5.2 на каждой итерации цикла изменяется не более 2 байт внутреннего состояния RC4, S: элементы $S[i]$ и $S[j]$, значения которых меняются местами. То есть если $i = 0$, $j = 4$ и $S[0] = 56$, $S[4] = 78$, то в результате операции обмена будем иметь $S[0] = 78$, $S[4] = 56$. Если j равно i, то $S[i]$ не изменяется.

Выглядит слишком просто, чтобы обеспечить безопасность, и тем не менее криптоаналитикам понадобилось 20 лет, чтобы найти дефекты, допускающие эксплуатацию. До обнаружения этих дефектов мы знали только о слабых местах RC4 в конкретных реализациях, в частности в первом стандарте шифрования Wi-Fi, WEP.

RC4 в WEP

WEP, протокол безопасности Wi-Fi первого поколения, в настоящее время полностью взломан из-за недостатков в конструкции протокола и в RC4.

В реализации WEP алгоритм RC4 применяется для шифрования полезной нагрузки кадров 802.11, дейтаграмм (или пакетов), транспортирующих данные по беспроводной сети. Вся полезная нагрузка доставляется в одном сеансе с применением одного и того же секретного ключа длиной 40 или 104 бита, но имеет предположительно уникальное 3-байтовое одноразовое число, являющееся частью заголовка кадра (той части кадра, в которой хранятся метаданные и которая предшествует самой полезной нагрузке). Видите, в чем проблема?

Проблема в том, что RC4 не поддерживает одноразовых чисел, по крайней мере о них нет ни слова в официальной спецификации, а потоковый шифр нельзя использовать без одноразового числа. Проектировщики WEP обошли это ограничение следующим образом: включили 24-битовое одноразовое число в заголовок беспроводного кадра и дописали его в начало ключа WEP, предполагая использовать как секретный ключ RC4. То есть если одноразовое число занимает байты $N[0]$, $N[1]$, $N[2]$, а ключ WEP – байты $K[0]$, $K[1]$, $K[2]$, $K[3]$, $K[4]$, то сам ключ RC4 будет равен $N[0]$, $N[1]$, $N[2]$, $K[0]$, $K[1]$, $K[2]$, $K[3]$, $K[4]$. Предполагалось, что 40-битовые секретные ключи должны давать 64-битовые эффективные ключи, а 104-битовые ключи – 128-битовые эффективные ключи. А что на самом деле? Объявленный 128-битовым протокол WEP фактически обеспечивает только 104-битовую безопасность, в лучшем случае.

Но у этого трюка с одноразовыми числами в WEP есть и реальные проблемы.

- **24 бита – слишком мало для одноразовых чисел.** Это означает, что если одноразовое число случайно выбирается для каждого нового сообщения, то спустя приблизительно $2^{24/2} = 2^{12}$ пакетов, или всего несколько мегабайтов трафика, мы встретим два пакета, зашифрованных с одним и тем же одноразовым числом и, стало быть, имеющих одинаковую гамму. Даже если одноразовое число – счетчик, пробегающий значения от 0 до $2^{24} - 1$, повторение случится после передачи нескольких гигабайтов данных, и тогда повторившееся случайное число позволит противнику дешифрировать пакеты. Но есть и более серьезная проблема.
- **Комбинирование одноразового числа с ключом описанным выше способом позволяет определить ключ.** Три несекретных байта одноразового числа в WEP открывают возможность найти

значение *S* после трех итераций алгоритма развертки ключа. Из-за этого криптоаналитики обнаружили, что первый байт гаммы сильно зависит от первого секретного байта ключа – четвертого байта, вставленного KSA, – и что это статистическое смещение можно использовать для получения секретного ключа.

Для эксплуатации этих недостатков необходим доступ к шифртекстам и гамме, т. е. известные или подобранные открытые тексты. Но это достаточно просто: известный открытый текст возникает, когда кадры Wi-Fi инкапсулируют данные с известным заголовком, а подобранные открытые тексты – когда противник внедряет известный открытый текст, зашифрованный ключом-мишенью. Таким образом, атака работает практически, а не только на бумаге.

После появления первых атак на WEP в 2001 году исследователи нашли более быстрые атаки, требующие меньше шифртекста. В настоящее время можно даже найти инструменты, например aircrack-ng, которые реализуют атаку целиком, от прослушивания сети до криптоанализа.

Небезопасность WEP вызвана как недостатками RC4, который принимает только ключ, а не ключ и одноразовое число (как любой достойный потоковый шифр), так и недостатками при проектировании самого WEP.

Теперь рассмотрим второй крупный «облом» RC4.

RC4 в TLS

TLS – самый важный протокол безопасности в интернете. Наиболее широкую известность ему принесло использование для установления HTTPS-соединений, но он применяется также для защиты некоторых виртуальных частных сетей (VPN), почтовых серверов, мобильных приложений и многого другого. И как это ни печально, TLS долгое время поддерживал RC4.

В отличие от WEP, в реализации TLS нет такой же вопиющей ошибки – манипулирования спецификацией RC4 с целью использовать открытое одноразовое число. Вместо этого TLS передает RC4 уникальный 128-битовый сеансовый ключ, т. е. в отличие от WEP в нем нет бросающихся в глаза недостатков.

Слабость TLS вызвана только неустранимыми дефектами RC4: статистическим смещением, или неслучайностью, что, как мы знаем, является поводом отказаться от потокового шифра. Например, второй байт гаммы, порождаемой RC4, является нулевым с вероятностью 1/128, тогда как в идеале вероятность должна быть равна 1/256. (Напомним, что байт может принимать 256 значений от 0 до 255, следовательно, истинно случайный байт должен быть равен 0 с вероятностью 1/256.) Поразительно, но большинство экспертов продолжали доверять RC4 аж до 2013 года, хотя о его статистической смещенности было известно с 2001 года.

Известия о статистической смещенности RC4 должно было быть достаточно, чтобы отправить этот шифр на свалку истории, даже если

бы мы не знали о том, как эксплуатировать эту смещенность для компрометации реальных приложений. В TLS дефекты RC4 не подвергались публичной эксплуатации до 2011 года, но предполагается, что АНБ использовало его недостатки для компрометации соединений, установленных с применением RC4, задолго до того.

Выяснилось, что смещен не только второй байт гаммы RC4, но и все первые 256 байт. Как обнаружилось в 2011 году, вероятность того, что один из этих байтов нулевой, равна $1/256 + c/256^2$, где константа c принимает значение от 0.24 до 1.34. Это верно не только для нуля, но и для других значений байтов. Удивительно, что RC4 терпит неудачу там, где успешны даже криптографически нестойкие PRNG, – в порождении равномерно распределенных псевдослучайных байтов (когда вероятность появления каждого из 256 байт равна 1/256).

Чтобы эксплуатировать основанную на дефектном RC4 реализацию TLS, достаточно даже самой слабой модели атаки: по существу, мы набираем шифртексты и ищем сам открытый текст, а не ключ. Но есть и подводный камень: нужно много шифртекстов, зашифровывающих *один и тот же открытый текст* с разными секретными ключами. Такую модель атаки иногда называют *широковещательной*, потому что она сродни широковещательной передаче одного и того же сообщения нескольким получателям.

Например, предположим, что мы хотим дешифрировать байт открытого текста P_1, имея много байтов, полученных перехватом различных шифртекстов для одного и того же сообщения. Первые четыре байта шифртекста будут тогда выглядеть следующим образом:

$$C_1^2 = P_1 \oplus KS_1^2;$$
$$C_1^3 = P_1 \oplus KS_1^3;$$
$$C_1^4 = P_1 \oplus KS_1^4.$$

Из-за смещенности RC4 байты гаммы KS_1^i с большей вероятностью равны нулю, чем любому другому значению. Поэтому байты C_1^i с большей вероятностью равны P_1, чем какому-то другому значению. Чтобы определить P_1, зная байты C_1^i, мы должны просто подсчитать, сколько раз встречается каждое значение байта, и вернуть в качестве P_1 самое частое. Но поскольку статистическое смещение очень мало, для мало-мальски достоверных выводов необходимы миллионы значений.

Атаку можно обобщить с целью восстановления более одного байта открытого текста и эксплуатировать не только одно смещенное значение (в данном случае нуль). Просто алгоритм становится чуть более сложным. Но провести такую атаку практически затруднительно, т. е. требуется набрать много шифртекстов, соответствующих одному открытому тексту, зашифрованному разными ключами. Например, эта атака не может вскрыть все защищенные TLS соединения, в которых используется RC4, потому что нужно как-то убедить сервер отправить один и тот же открытый текст, зашифрованный разными ключами, многим разным получателям или много раз одному получателю.

Salsa20

Salsa20 – простой программный шифр, оптимизированный для современных процессоров. Он, а также его вариант ChaCha применяются в различных протоколах и библиотеках. Спроектировавший его авторитетный криптограф Дэниел Дж. Бернштейн представил Salsa20 на конкурс eSTREAM в 2005 году. Шифр был включен в портфель ПО, отобранного по результатам конкурса. Благодаря своей простоте и скорости работы Salsa20 завоевал популярность у разработчиков.

Salsa20 – потоковый шифр на основе счетчика; он генерирует гамму, повторно обрабатывая счетчик, который увеличивается для каждого нового блока. Как видно на рис. 5.10, *базовый* алгоритм Salsa20 преобразует 512-байтовый блок с помощью ключа (*K*), одноразового числа (*N*) и значения (*Ctr*). Затем Salsa20 прибавляет результат к первоначальному значению блока для порождения *блока гаммы*. (Если бы алгоритм возвращал перестановку, сгенерированную базовым алгоритмом, непосредственно, то был бы абсолютно небезопасным, потому что его можно было бы обратить. Финальное прибавление начального секретного состояния $K||N||Ctr$ делает преобразование ключа в блок гаммы необратимым.)

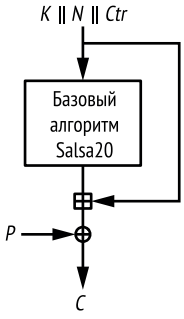

Рис. 5.10. Схема шифрования 512-битового блока открытого текста в алгоритме Salsa20

Функция quarterround

В базовом алгоритме Salsa20 используется функция *quarterround* (**QR**), которая преобразует 32-битовые слова (*a*, *b*, *c*, *d*), как показано ниже:

$$b = b \oplus [(a + d) <<< 7];$$
$$c = c \oplus [(b + a) <<< 9];$$
$$d = d \oplus [(c + b) <<< 13];$$
$$a = a \oplus [(d + c) <<< 18].$$

Эти четыре выражения вычисляются сверху вниз, т. е. новое значение *b* зависит от *a* и *d*, новое значение *c* – от *a* и нового значения *b* (и, следовательно, также от *d*) и так далее.

Символом <<< обозначена операция циклического сдвига слова влево на указанное число битов, которое может принимать значение

от 1 до 31 (для 32-битовых слов). Например, <<< 8 сдвигает биты слова на восемь позиций влево, как показано в следующих примерах:

0x01234567 <<< 8 = 0x23456701;
0x01234567 <<< 16 = 0x45670123;
0x01234567 <<< 22 = 0x59c048d1.

Преобразование 512-битового состояния в Salsa20

Базовая перестановка в Salsa20 преобразует 512-битовое внутреннее состояние, рассматриваемое как массив 32-битовых слов 4×4. На рис. 5.11 показано начальное состояние, состоящее из восьми слов ключа (256 бит), двух слов одноразового числа (64 бита), двух слов счетчика (64 бита) и четырех фиксированных слов (128 бит), одинаковых для любого акта шифрования и дешифрирования и для всех блоков.

Для преобразования начального 512-битового состояния Salsa20 сначала применяет преобразование **QR** ко всем четырем столбцам независимо (так называемый *столбцовый раунд*), а затем ко всем четырем строкам независимо (*строковый раунд*), как показано на рис. 5.12. Последовательность, состоящая из столбцового и строкового раундов, называется *двойным раундом*. Salsa20 выполняет 10 двойных раундов, т. е. всего 20 раундов, что объясняет появление числа *20* в названии *Salsa20*.

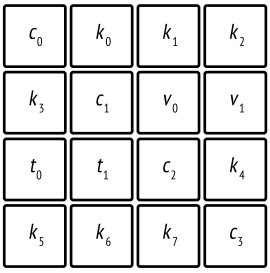

Рис. 5.11. Инициализация состояния Salsa20

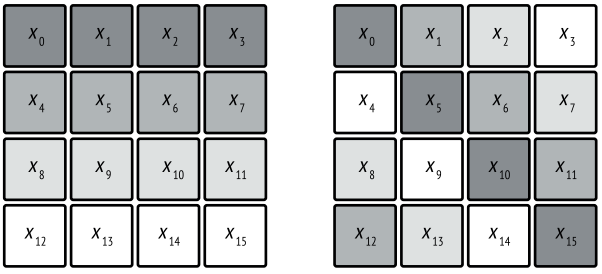

*Рис. 5.12. Столбцы и строки, преобразуемые функцией quarterround (**QR**)*

В **столбцовом раунде** четыре столбца преобразуются следующим образом:

$$\mathbf{QR}(x_0, x_4, x_8, x_{12});$$
$$\mathbf{QR}(x_1, x_5, x_9, x_{13});$$
$$\mathbf{QR}(x_2, x_6, x_{10}, x_{14});$$
$$\mathbf{QR}(x_3, x_7, x_{11}, x_{15}).$$

В **строковом раунде** строки преобразуются следующим образом:

$$\mathbf{QR}(x_0, x_1, x_2, x_3);$$
$$\mathbf{QR}(x_5, x_6, x_7, x_4);$$
$$\mathbf{QR}(x_{10}, x_{11}, x_8, x_9);$$
$$\mathbf{QR}(x_{15}, x_{12}, x_{13}, x_{14}).$$

Заметим, что в столбцовом раунде каждый вызов **QR** принимает аргументы x_i, упорядоченные сверху вниз, а в строковом раунде первыми аргументами **QR** являются слова, расположенные на диагонали (как показано в массиве справа на рис. 5.12), а не слова в первом столбце.

Вычисление Salsa20

В листинге 5.3 показаны начальные состояния первого и второго блоков Salsa20, когда ключ состоит из одних нулей (байтов `00`), а одноразовое число – из одних единиц (байтов `ff`). Эти состояния отличаются только одним битом счетчика, выделенного жирным шрифтом; он равен 0 в первом блоке и 1 во втором.

Листинг 5.3. Начальные состояния первых двух блоков Salsa20, когда ключ состоит из одних нулей, а одноразовое число – из одних единиц

```
61707865 00000000 00000000 00000000     61707865 00000000 00000000 00000000
00000000 3320646e ffffffff ffffffff     00000000 3320646e ffffffff ffffffff
00000000 00000000 79622d32 00000000     00000001 00000000 79622d32 00000000
00000000 00000000 00000000 6b206574     00000000 00000000 00000000 6b206574
```

Но, несмотря на различие всего в одном бите, соответствующие внутренние состояния после 10 двойных раундов не имеют ничего общего, как видно из листинга 5.4.

Листинг 5.4. Состояния, показанные в листинге 5.3, после 10 двойных раундов Salsa20

```
e98680bc f730ba7a 38663ce0 5f376d93     1ba4d492 c14270c3 9fb05306 ff808c64
85683b75 a56ca873 26501592 64144b6d     b49a4100 f5d8fbbd 614234a0 e20663d1
6dcb46fd 58178f93 8cf54cfe cfdc27d7     12e1e116 6a61bc8f 86f01bcb 2efead4a
68bbe09e 17b403a1 38aa1f27 54323fe0     77775a13 d17b99d5 eb773f5b 2c3a5e7d
```

Напомним, однако, что какими бы случайными ни казались значения слов в блоке гаммы, это еще далеко не гарантия безопасности.

Выход RC4 выглядит случайным, но в нем присутствует роковое статистическое смещение. По счастью, Salsa20 гораздо безопаснее RC4, и никакого статистического смещения в нем нет.

Дифференциальный криптоанализ

Чтобы продемонстрировать, почему Salsa20 безопаснее RC4, рассмотрим основы *дифференциального криптоанализа*, который изучает различия между состояниями, а не их фактические значения. Например, два начальных состояния на рис. 5.13 отличаются одним битом счетчика, или словом *x*8 в массиве состояния Salsa20. Их поразрядное различие показано в массиве ниже:

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

Различие между двумя состояниями описывается результатом применения к ним операции XOR. Бит 1, выделенный полужирным шрифтом, соответствует различию в одном бите между двумя состояниями. Если применить к обоим состояниям XOR, то различиям будут соответствовать позиции результата, содержащие 1.

Чтобы увидеть, насколько быстро распространяются изменения начального состояния в результате выполнения базового алгоритма Salsa20, посмотрим на различия между состояниями в процессе повторения раундов. После одного раунда различие распространяется на два из трех слов в первом столбце:

```
80040003 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00002000 00000000 00000000 00000000
```

После двух раундов различия распространяются на уже различающиеся строки, т. е. на все, кроме второй. В этот момент различия между состояниями еще относительно разрежены; изменилось не так много битов в слове:

```
9ed7eb7f 060002c0 18028b0c 57ca83c0
00000000 00000000 00000000 00000000
00000001 0000e000 801c0006 00000000
00002000 00400000 04000008 0060f300
```

После трех раундов различия становятся более плотными, хотя наличие большого числа нулевых полубайтов указывает, что многие позиции еще не затронуты начальным различием:

```
3ab3c25d 9f40a5c9 10070e30 07bd03c0
db1ee2ce 43ee9401 21a702c3 48fd800c
403c1e72 00034003 4dc843be 700b8857
5625b75b 09c00e00 06000348 23f712d4
```

После четырех раундов различия кажутся человеку случайными, и статистический анализ тоже показывает, что они почти случайны:

```
d93bed6d a267bf47 760c2f9f 4a41d54b
0e03d792 7340e010 119e6a00 e90186af
7fa9617e b6aca0d7 4f6e9a4a 564b34fd
98be796d 64908d32 4897f7ca a684a2df
```

Итак, понадобилось всего четыре раунда, чтобы различие в одном бите распространилось на большую часть 512-битового состояния. В криптографии это называется *полной диффузией*.

Мы видели, что в раундах Salsa20 различия распространяются быстро. Но мало того, что различия распространяются по всему состоянию, они еще и описываются сложными уравнениями, которые затрудняют предсказания будущих различий из-за своей *нелинейности*, обусловленной смесью операций XOR, сложения и циклического сдвига. Если бы использовались только операции XOR, то различия также распространялись бы быстро, но процесс оказался бы линейным, а потому небезопасным.

Атака на Salsa20/8

В Salsa20 по умолчанию выполняется 20 раундов, но иногда для ускорения используется всего 12 раундов – эта версия называется Salsa20/12. Хотя в Salsa20/12 на восемь раундов меньше, чем в Salsa20, он все равно криптографически гораздо более стойкий, чем ослабленный вариант Salsa20/8 с восьмью раундами, который используется реже.

Для взлома Salsa20 в идеале понадобилось бы 2^{256} операций, поскольку ключ 256-битовый. Если ключ можно восстановить, выполнив меньше 2^{256} операций, то теоретически шифр можно считать взломанным. Именно так обстоит дело с Salsa20/8.

Атака на Salsa20/8 (опубликованная в 2008 году в статье «New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba», соавтором которой я являюсь и которая получила приз за криптоанализ от Дэниеля Дж. Бернштейна) эксплуатирует статистическое смещение в базовом алгоритме Salsa после четырех раундов для нахождения ключа восьмираундового Salsa20. На самом деле эта атака носит в основном теоретический характер: мы оценили ее сложность как 2^{251} вычислений базовой функции – это все равно невозможно, хотя и меньше ожидаемой сложности 2^{256} операций.

В атаке эксплуатируется не только смещение первых четырех раундов Salsa20/8, но и свойство последних четырех раундов: если

известны одноразовое число *N* и счетчик *Ctr* (см. рис. 5.10), то для обращения вычисления гаммы и получения начального состояния нужен только ключ *K*. Но, как показано на рис. 5.13, если известна лишь какая-то часть *K*, то можно частично обратить вычисление до четвертого раунда и наблюдать некоторые биты этого промежуточного состояния – включая смещенный бит! Увидеть смещение можно, только если правильно угадан частичный ключ, поэтому наличие смещения – признак того, что ключ правилен.

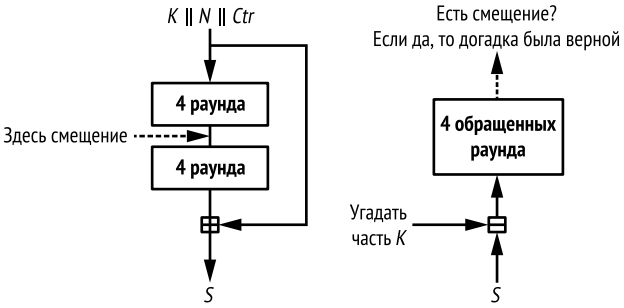

Рис. 5.13. Принцип атаки на Salsa20/8

В реальной атаке на Salsa20/8, чтобы определить, верна ли догадка, нужно угадать 220 бит ключа и понадобится 2^{31} пар блоков гаммы, все с одинаковым конкретным различием в одноразовом числе. Выделив 220 правильных бит, остальные 36 бит можно найти полным перебором. Полный перебор требует 2^{36} операций – вычисление, мизерное по сравнению с неосуществимыми $2^{220} \times 2^{31} = 2^{251}$ попытками, необходимыми для определения 220 бит при выполнении первой части атаки.

Какие возможны проблемы

К сожалению, потоковые шифры подвержены различным проблемам, от хрупкой и небезопасной конструкции до неправильной реализации криптостойких алгоритмов. В следующих разделах я рассмотрю примеры потенциальных проблем из каждой категории.

Повторное использование одноразового числа

Самая распространенная ошибка в потоковых шифрах – чисто любительская: она возникает, когда одноразовое число используется более одного раза с одним и тем же ключом. В этом случае получаются одинаковые гаммы, что позволяет взломать шифр, применив к двум шифртекстам операцию XOR. Тогда гамма взаимно уничтожается, и остается результат XOR двух открытых текстов.

Например, в старых версиях Microsoft Word и Excel использовалось уникальное одноразовое число для каждого документа, но это число не изменялось при модификации документа. В результате откры-

тый и зашифрованный текст старой версии документа можно было использовать для дешифрирования более поздних зашифрованных версий. Если даже Microsoft допустила такую ошибку, то можете представить себе масштаб проблемы.

Некоторые потоковые шифры, спроектированные в 2010-х годах, пытались уменьшить риск повторного использования одноразового числа путем создания «устойчивых к неправильному применению» конструкций или шифров, остающихся безопасными даже при повторном использовании одноразового числа. Однако такой уровень безопасности сопряжен с потерей производительности, как мы увидим в главе 8 при обсуждении режима SIV.

Дефектная реализация RC4

И так уже слабый шифр RC4 может стать еще слабее в результате непродуманной оптимизации его реализации. Например, рассмотрим следующее предложение, поданное в 2007 году на конкурс Underhanded C – неформальное состязание программистов по написанию безвредного, на первый взгляд, кода, который на самом деле включает вредоносную функцию.

Вот как это работает. Наивный способ реализации строки `swap(S[i], S[j])` в алгоритме RC4 выглядит (на Python) следующим образом:

```
buf = S[i]
S[i] = S[j]
S[j] = buf
```

Этот способ обмена двух переменных, безусловно, работает, но нужно создавать новую переменную `buf`. Чтобы избежать этого, программисты часто применяют показанный ниже трюк *XORswap* для обмена значений переменных `x` и `y`:

```
x = x ⊕ y
y = x ⊕ y
x = x ⊕ y
```

Этот трюк работает, потому что во второй строке `y` присваивается значение $x \oplus y \oplus y = x$, а в третьей строке `x` получает значение $x \oplus y \oplus x \oplus y \oplus y = y$. Используя этот трюк в реализации RC4, мы получаем код, показанный в листинге 5.5 (немного адаптированный код из программы Вагнера и Бьонди, поданной на конкурс Underhanded C и опубликованной по адресу *http://www.underhanded-c.org/_page_id_16.html*).

Листинг 5.5. Неправильная реализация на C шифра RC4, ошибка связана с использованием трюка XORswap

```
# define TOBYTE(x) (x) & 255
# define SWAP(x,y) do { x^=y; y^=x; x^=y; } while (0)
```

```
static unsigned char S[256];
static int i=0, j=0;

void init(char *passphrase) {
    int passlen = strlen(passphrase);
    for (i=0; i<256; i++)
        S[i] = i;
    for (i=0; i<256; i++) {
        j = TOBYTE(j + S[TOBYTE(i)] + passphrase[j % passlen]);
        SWAP(S[TOBYTE(i)], S[j]);
    }
    i = 0; j = 0;
}

unsigned char encrypt_one_byte(unsigned char c) {
    int k;
    i = TOBYTE(i+1);
    j = TOBYTE(j + S[i]);
    SWAP(S[i], S[j]);
    k = TOBYTE(S[i] + S[j]);
    return c ^ S[k];
}
```

Теперь сделайте паузу и попытайтесь найти в листинге 5.5 проблему, связанную с обменом посредством XOR.

Беда происходит, когда `i = j`. Вместо того чтобы оставить состояние неизменным, обмен посредством XOR присваивает `S[i]` значение $S[i] \oplus S[i] = 0$. В результате байт состояния будет обнулен каждый раз, как `i` оказывается равно `j` в алгоритме развертки ключа или в процессе шифрования, и в конечном итоге состояние, а значит, и гамма будут состоять из одних нулей. Например, после обработки 68 КБ данных большая часть байтов 256-байтового состояния равна нулю, а выходная гамма выглядит следующим образом:

```
00 00 00 00 00 00 00 53 53 00 00 00 00 00 00 00 00 00 00 00 13 13 00 5c 00 a5 00 00 …
```

Урок такой – не нужно чрезмерно оптимизировать реализации криптографических алгоритмов. В криптографии понятность и уверенность всегда важнее производительности.

Слабые аппаратно реализованные шифры

Когда выясняется, что криптосистема небезопасна, некоторые системы способны быстро отреагировать, ненавязчиво обновив программное обеспечение дистанционно (как в некоторых системах платного телевидения) или выпустив новую версию и уведомив пользователей о необходимости выполнить обновление (как в случае мобильных приложений). Но старым системам не так повезло, и они вынуждены некоторое время работать со скомпрометированной криптосистемой до перехода на безопасную версию, как в случае некоторых спутниковых телефонов.

В начале 2000-х годов учреждения стандартизации связи в США и ЕС (TIA и ETSI) совместно разработали два стандарта спутниковой связи. Спутниковые телефоны похожи на мобильные, но отличаются тем, что сигнал проходит через спутники, а не через наземные станции. Преимущество в том, что использовать их можно практически везде, а недостатки – цена, качество связи, задержка и, как оказалось, безопасность.

GMR-1 и GMR-2 – два стандарта спутниковой телефонии, принятых большинством коммерческих производителей, в т. ч. Thuraya и Inmarsat. В оба включены потоковые шифры для шифрования голосовой связи. Шифр в GMR-1 аппаратный и представляет собой комбинацию четырех LFSR; он похож на A5/2, преднамеренно небезопасный шифр в стандарте мобильной связи 2G, ориентированном на незападные страны. Шифр в GMR-2 программный, с 8-байтовым состоянием и S-блоками. Оба потоковых шифра небезопасны и защищают только от любителей, но не от государственных органов.

Эта история призвана напомнить, что потоковые шифры раньше было проще взломать, чем блочные, и что они более уязвимы для подрывных действий. Почему? Да потому, что если вы специально проектируете слабый шифр, то при обнаружении дефекта сможете сослаться на общую нестойкость потоковых шифров, не признаваясь в дурных намерениях.

Для дополнительного чтения

Если вы хотите больше узнать о потоковых шифрах, начните с архивов конкурса eSTREAM по адресу *http://www.ecrypt.eu.org/stream/project.html*. Там вы найдете сотни статей на тему потоковых шифров, в т. ч. подробное описание более 30 кандидатов и многих атак. К числу наиболее интересных относятся корреляционные, алгебраические и кубические атаки. См., в частности, работу Куртуа (Courtois) и Мейера (Meier) по первым двум типам атак и работу Динура (Dinur) и Шамира (Shamir) по кубическим атакам.

Дополнительные сведения о RC4 см. в работе Патерсона с сотрудниками по адресу *http://www.isg.rhul.ac.uk/tls/*; в ней обсуждается безопасность использования RC4 в TLS и WPA. Также см. описание шифра Spritz, похожего на RC4 и спроектированного в 2014 году Ривестом, тем самым, который создал RC4 в 1980-е годы.

Наследие Salsa20 также заслуживает внимания. Потоковый шифр ChaCha, похожий на Salsa20, но с немного другой базовой перестановкой, впоследствии использовался в функции хеширования BLAKE, с которой мы познакомимся в главе 6. Все эти алгоритмы используют методы программной реализации Salsa20 с применением распараллеленных команд (см. *https://cr.yp.to/snuffle.html*).

6

ФУНКЦИИ ХЕШИРОВАНИЯ

Функции хеширования, или хеш-функции, например MD5, SHA-1, SHA-256, SHA-3 и BLAKE2, – швейцарский нож криптографа; они используются для цифровой подписи, шифрования с открытым ключом, проверки целостности, аутентификации сообщений, защиты паролей, в протоколах выработки ключей и во многих других криптографических протоколах. Не важно, зашифровываете вы письмо, отправляете сообщение с помощью мобильного телефона, заходите на веб-сайт по протоколу HTTPS или подключаетесь к удаленной машине по протоколу IPSec или SSH, где-то под капотом прячется хеш-функция.

Функции хеширования – безусловно, самый многогранный и вездесущий из криптографических алгоритмов. Можно привести множество примеров их реального использования: в облачных системах хранения они служат для нахождения одинаковых файлов и обнару-

жения модифицированных файлов; в системе управления версиями Git – для идентификации файлов, хранящихся в репозитории; в хостовых системах обнаружения вторжений (host-based intrusion detection systems – HIDS) – для обнаружения модифицированных файлов; в сетевых системах обнаружения вторжений (network-based intrusion detection systems – NIDS) – для обнаружения заведомо вредоносных данных, перемещающихся по сети; в компьютерно-технической экспертизе – для доказательства того, что цифровые артефакты не были модифицированы; в технологии биткойна – в системах доказательства проделанной работы. Приложений не счесть.

В отличие от потоковых шифров, которые создают длинный выход по короткому входу, хеш-функции принимают длинный вход и формируют короткий выход, называемый *хеш-значением*, или *дайджестом* (см. рис. 6.1).

Рис. 6.1. Вход и выход хеш-функции

Эта глава посвящена двум основным темам. Первая – безопасность: что такое безопасная функция? В этой связи я введу два аспекта безопасности: стойкость к коллизиям и стойкость к восстановлению прообраза. Вторая большая тема – построение хеш-функций. Мы дадим общий обзор методов, применяемых в современных хеш-функциях, а затем рассмотрим внутреннее устройство наиболее распространенных из них: SHA-1, SHA-2, SHA-3 и BLAKE2. Наконец, мы увидим, как безопасные хеш-функции могут вести себя небезопасно при неправильном применении.

Примечание *Не путайте криптографические хеш-функции с некриптографическими. Некриптографические хеш-функции используются в структурах данных, например в хеш-таблицах, а также для обнаружения непреднамеренных ошибок, они не претендуют ни на какую безопасность. Например, код циклической избыточности (CRC) – это некриптографическая хеш-функция, применяемая для обнаружения случайных модификаций файла.*

Безопасные хеш-функции

Понятие безопасности для хеш-функций отличается от того, что мы видели до сих пор. Если шифры защищают конфиденциальность данных, стремясь гарантировать, что данные, отправленные по открытым каналам связи, невозможно прочитать, то хеш-функции защищают целостность данных, стремясь гарантировать, что данные – открытые или зашифрованные – не были модифицированы. Если хеш-функция безопасна, то хеш-значения двух разных сообщений будут различны. Поэтому хеш-значение файла может служить его идентификатором.

Рассмотрим самое распространенное применение хеш-функции: *цифровые подписи*, или просто *подписи*. При использовании цифровых подписей приложение обрабатывает хеш подписываемого сообщения, а не само сообщение, как показано на рис. 6.2. Хеш выступает в роли идентификатора сообщения. Если изменить всего один бит сообщения, то его хеш станет совершенно другим. Поэтому хеш-функция служит гарантией того, что сообщение не было модифицировано. Подписание хеша сообщения столь же безопасно, как подписание самого сообщения, а подписать короткий хеш, скажем состоящий из 256 бит, гораздо быстрее, чем полное сообщение, которое может быть очень длинным. На самом деле большинство алгоритмов цифровой подписи могут работать только с короткими входными данными типа хеш-значений.

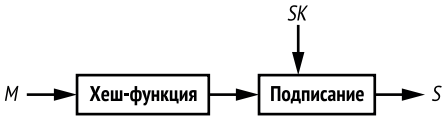

Рис. 6.2. Хеш-функция в схеме цифровой подписи. Хеш играет роль заместителя сообщения

И снова непредсказуемость

Криптографическая стойкость хеш-функций проистекает из непредсказуемости их выхода. Взять 256-битовые шестнадцатеричные значения, показанные ниже, – эти хеши вычислены с помощью описанной в стандарте NIST функции хеширования SHA-256, которой на вход подавались буквы `a`, `b` и `c`. Как видим, хотя значения `a`, `b` и `c` в кодировке ASCII отличаются только одним или двумя битами (`a` имеет код 01100001, `b` – 01100010, `c` – 01100011), их хеш-значения совершенно различны.

```
SHA-256("a") = ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
SHA-256("b") = 3e23e8160039594a33894f6564e1b1348bbd7a0088d42c4acb73eeaed59c009d
SHA-256("c") = 2e7d2c03a9507ae265ecf5b5356885a53393a2029d241394997265a1a25aefc6
```

Зная только эти три хеша, невозможно предсказать результат применения SHA-256 к `d` или любому из его битов. Почему? Потому что хеш-значения безопасной хеш-функции *непредсказуемы*. Безопасная хеш-функция должна выглядеть как черный ящик, возвращающий случайную строку всякий раз, как получает какие-то данные.

Общее теоретическое определение безопасной хеш-функции говорит, что она ведет себя как истинно случайная функция (иногда называемая *случайным оракулом*). Точнее, случайная хеш-функция не должна обладать никаким свойством или закономерностью, которым не обладала бы случайная функция. Это определение полезно теоретикам, но на практике нам нужны более специфические понятия, а именно стойкость к восстановлению прообраза и стойкость к коллизиям.

Стойкость к восстановлению прообраза

Прообразом данного хеш-значения H является любое сообщение M такое, что **Hash**(M) = H. Стойкость к восстановлению прообраза означает, что противник, знающий случайное хеш-значение, не сможет найти его прообраз. Иногда хеш-функции называют *односторонними функциями*, потому что перейти от сообщения к хешу легко, а в обратную сторону невозможно.

Прежде всего отметим, что хеш-функцию невозможно обратить даже при наличии неограниченной вычислительной мощности. Предположим, к примеру, что я хешировал некоторое сообщение функцией SHA-256 и получил такое 256-битовое хеш-значение:

```
f67a58184cef99d6dfc3045f08645e844f2837ee4bfcc6c949c9f7674367adfd
```

Даже при наличии неограниченной вычислительной мощности вы никогда не сможете определить, *какое именно* сообщение породило этот хеш, потому что существует много сообщений с одинаковым хешем. Вы могли бы найти *какие-то* сообщения, порождающие данное хеш-значение (и, возможно, среди них будет выбранное мной), но никогда не сможете точно сказать, какое сообщение я выбрал.

Например, существует 2^{256} возможных значений 256-битового хеша (типичная длина практически используемых хеш-функций), но гораздо больше, скажем, 1024-битовых сообщений (а именно 2^{1024} возможных значений). Отсюда следует, что в среднем у каждого из возможных 256-битовых хеш-значений будет $2^{1024} / 2^{256} = 2^{1024 - 256} = 2^{768}$ прообразов длиной 1024 бита.

На практике мы должны быть уверены в практической невозможности найти *хотя бы одно* сообщение, отображаемое на данное хеш-значение, а не только сообщение, которое фактически было использовано, именно это понимается под стойкостью к восстановлению прообраза. Точнее, речь может идти о стойкости к восстановлению первого и второго прообразов. *Стойкость к восстановлению первого прообраза* (или просто *стойкость к восстановлению прообраза*) означает, что практически невозможно найти сообщение, имеющее данное хеш-значение. А *стойкость к восстановлению второго прообраза* означает, что для данного сообщения M_1 практически невозможно найти другое сообщение M_2 с таким же хеш-значением.

Стоимость восстановления прообразов

Если известны хеш-функция и хеш-значение, то первые прообразы можно поискать, проверяя различные сообщения, пока не будет найдено сообщение с таким же хеш-значением. Для этого служит алгоритм типа `find-preimage()` в листинге 6.1.

В листинге 6.1 функция `random_message()` генерирует случайное сообщение (скажем, 1024-битовое). Очевидно, что `find-preimage()` никогда не завершится, если длина хеша в битах, n, достаточно велика, потому что для нахождения прообраза нужно в среднем 2^n попыток.

Это безнадежно в случае $n = 256$, как в современных хеш-функциях типа SHA-256 или BLAKE2.

Листинг 6.1. Оптимальный алгоритм поиска прообраза для безопасной хеш-функции

```
find-preimage(H) {
    repeat {
        M = random_message()
        if Hash(M) == H then return M
    }
}
```

Почему стойкость к восстановлению второго прообраза ниже

Я утверждаю, что если можно найти первые прообразы, то можно найти и вторые прообразы (для той же хеш-функции). Действительно, если алгоритм `solve-preimage()` возвращает прообраз заданного хеш-значения, то его можно использовать для нахождения второго прообраза некоторого сообщения *M*, как показано в листинге 6.2.

Листинг 6.2. Как найти вторые прообразы, если мы умеем находить первые прообразы

```
solve-second-preimage(M) {
    H = Hash(M)
    return solve-preimage(H)
}
```

Таким образом, мы сможем найти второй прообраз, рассматривая это как задачу восстановления прообраза и применив атаку на прообраз. Отсюда следует, что любая хеш-функция, стойкая к восстановлению второго прообраза, является также стойкой к восстановлению прообраза. (Иначе она не была бы стойкой к восстановлению второго прообраза, как показывает приведенный выше алгоритм `solve-second-preimage`.) Иными словами, наилучшая атака, которую можно применить для нахождения вторых прообразов, почти идентична наилучшей атаке для нахождения первых прообразов (если только у хеш-функции нет дефекта, открывающего возможность для более эффективных атак). Заметим также, что атака поиска прообраза, по сути дела, совпадает с атакой восстановления ключа на блочный или потоковый шифр – полный перебор в поисках одного-единственного значения.

Стойкость к коллизиям

Какую бы хеш-функцию ни выбрать, коллизии неизбежны вследствие принципа Дирихле, согласно которому если по *m* клеткам рассадить

n кроликов, то при $n > m$ по крайней мере в одной клетке окажется более одного кролика.

Примечание *Этот принцип можно обобщить и на другие предметы и контейнеры. Например, любая последовательность 27 слов в Конституции США содержит по крайней мере два слова, начинающихся с одной буквы. В мире хеш-функций клетками являются хеш-значения, а кроликами – сообщения. Поскольку мы знаем, что сообщений гораздо больше, чем хеш-значений, коллизии обязательно будут.*

Однако, вопреки очевидному, коллизии должно быть так же трудно найти, как исходное сообщение, только тогда хеш-функция будет считаться *стойкой к коллизиям*. Иными словами, противник не должен иметь возможность найти два разных сообщения с одинаковым хеш-значением.

Понятие стойкости к коллизиям связано с понятием стойкости к восстановлению второго прообраза: если для хеш-функции можно найти вторые прообразы, то можно найти и коллизии, как доказывает псевдокод в листинге 6.3.

Листинг 6.3. Наивный алгоритм поиска коллизий

```
solve-collision() {
    M = random_message()
    return (M, solve-second-preimage(M))
}
```

То есть любая стойкая к коллизиям хеш-функция является также стойкой к восстановлению второго прообраза. Если бы это было не так, то мы имели бы алгоритм восстановления второго прообраза, который позволил бы обойти стойкость к коллизиям.

Нахождение коллизий

Найти коллизии можно быстрее, чем прообразы, поскольку для этого требуется порядка $2^{n/2}$, а не 2^n операций вследствие *атаки на основе парадокса дней рождения*, идея которой заключается в следующем: если дано N сообщений и столько же хеш-значений, то можно породить $N \times (N - 1)/2$ потенциальных коллизий, рассматривая каждую *пару* хеш-значений (их количество имеет порядок N^2). Эта атака обычно иллюстрируется так называемым парадоксом дней рождений, согласно которому вероятность того, что в группе из 23 человек найдутся двое, родившихся в один день, равна 1/2.

Примечание *$N \times (N - 1)/2$ – это количество пар различных сообщений, которое мы делим на 2, потому что пары (M_1, M_2) и (M_2, M_1) считаются одинаковыми. Иными словами, порядок следования в паре нам безразличен.*

Для сравнения: в случае поиска прообраза N сообщений дают только N потенциальных прообразов, тогда как те же N сообщений дают приблизительно N^2 потенциальных коллизий. Поэтому мы говорим, что шансов найти решение *квадратично* больше. А сложность поиска, в свою очередь, оказывается квадратично меньше: чтобы найти коллизию, количество сообщений должно быть равно квадратному корню из 2^n, т. е. $2^{n/2}$.

Наивная атака на основе парадокса дней рождения

Вот простейший способ провести атаку на основе парадокса дней рождения для нахождения коллизий.

1. Вычислить $2^{n/2}$ хешей $2^{n/2}$ произвольно выбранных сообщений и сохранить все пары (сообщение, хеш) в списке.
2. Отсортировать список по хеш-значению, так чтобы одинаковые хеш-значения оказались рядом.
3. Поискать в отсортированном списке два соседних элемента с одинаковым хеш-значением.

К сожалению, этот метод требует очень много памяти (для хранения $2^{n/2}$ пар сообщение–хеш), а сортировка такого большого списка замедляет поиск, поскольку требует в среднем $n2^n$ операций даже при использовании алгоритма быстрой сортировки.

Поиск коллизий с низким потреблением памяти: ро-метод

Ро-метод – это алгоритм нахождения коллизий, который, в отличие от наивной атаки на основе парадокса дней рождения, требует небольшого объема памяти. Он работает следующим образом.

1. Для данной функции хеширования, порождающей n-битовые хеш-значения, выбрать случайным образом хеш-значение (H_1) и положить $H_1 = H'_1$.
2. Вычислить $H_2 = \mathbf{Hash}(H_1)$ и $H'_2 = \mathbf{Hash}(\mathbf{Hash}(H'_1))$; т. е. в первом случае мы применяем хеш-функцию один раз, а во втором дважды.
3. Повторяя этот процесс, вычислять $H_{i+1} = \mathbf{Hash}(H_i)$, $H'_{i+1} = \mathbf{Hash}(\mathbf{Hash}(H'_i))$, пока не найдется такое i, что $H_{i+1} = H'_{i+1}$.

Рисунок 6.3 помогает наглядно представить эту атаку. Стрелка, ведущая, например, из H_1 в H_2, означает, что $H_2 = \mathbf{Hash}(H_1)$. Заметим, что последовательность H_i рано или поздно зацикливается, напоминая по форме греческую букву ро (ρ). Цикл начинается в H_5 и характеризуется коллизией $\mathbf{Hash}(H_4) = \mathbf{Hash}(H_{10}) = H_5$. Ключевое наблюдение состоит в том, что для нахождения коллизии достаточно просто найти такой цикл. Описанный алгоритм позволяет противнику обнаружить цикл и, следовательно, найти коллизию.

Передовые методы поиска коллизий сначала находят начало цикла, а затем коллизию, не требуя хранить многочисленные значения в памяти и сортировать длинный список. Ро-метод для достижения

успеха выполняет приблизительно $2^{n/2}$ операций. На рис. 6.3 показано гораздо меньше хеш-значений, чем порождает реальная функция хеширования, генерирующая значения длиной 256 или более бит. В среднем цикл и хвост (часть от H_1 до H_5 на рис. 6.3) включают приблизительно по $2^{n/2}$ хеш-значений, где n – длина хеш-значения в битах. Поэтому для нахождения коллизии потребуется $2^{n/2} + 2^{n/2}$ раз вычислить функцию хеширования.

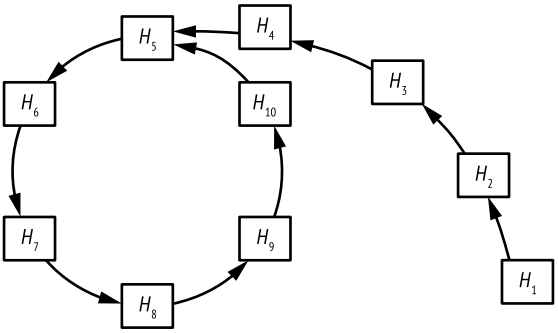

Рис. 6.3. Структура хеш-функции в ро-методе. Стрелками обозначены вычисления хеш-функции. Цикл, начинающийся в H_5, соответствует коллизии ***Hash***(H_4) = ***Hash***(H_{10}) = H

Построение функций хеширования

В 1980-х годах криптографы поняли, что простейший способ хешировать сообщение – разбить его на части и обработать каждую часть последовательно одним и тем же или похожим алгоритмом. У этой стратегии, называемой *итеративным хешированием*, есть две основные формы:

- итеративное хеширование с помощью *функции сжатия*, которая преобразует выход в *меньший выход*, как показано на рис. 6.4. Эта техника называется также построением *Меркла–Дамгора* (в честь криптографов Ральфа Меркла и Ивана Дамгора);
- итеративное хеширование с помощью функции, которая преобразует вход в выход *такого же размера*, но так, что различные входы дают различные выходы (*перестановка*), как показано на рис. 6.7. Такие функции называются *функциями губки*.

Мы обсудим, как на практике работают эти построения и как выглядят функции сжатия.

Хеш-функции на основе сжатия: построение Меркла–Дамгора

Все хеш-функции, разработанные с 1980-х по 2010-е годы, основаны на построении Меркла–Дамгора (М-Д): MD4, MD5, SHA-1 и семейство

SHA-2, а также менее известные RIPEMD и Whirlpool. Построение М-Д не идеальное, но простое и доказавшее достаточную безопасность во многих приложениях.

Примечание *В MD4, MD5 и RIPEMD буквы MD означают «message digest» (дайджест сообщения), а не Merkle–Damgård.*

Для хеширования сообщения построение М-Д разбивает сообщение на блоки одинаковой длины и перемешивает эти блоки с внутренним состоянием, применяя функцию сжатия, как показано на рис. 6.4. Здесь H_0 – начальное внутреннее состояние (обозначаемое IV), значения H_1, H_2, ... называются *цепными значениями*, а конечное внутреннее состояние является хеш-значением сообщения.

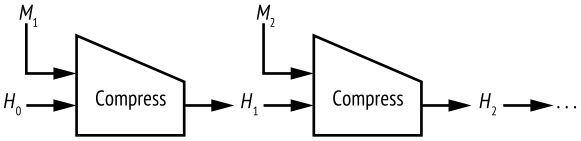

Рис. 6.4. Построение Меркла–Дамгора с использованием функции сжатия Compress

Блоки сообщения обычно имеют длину 512 или 1024 бита, но в принципе могут быть любого размера. Однако длина блока для данной функции хеширования фиксирована. Например, SHA-256 работает с 512-битовыми блоками, а SHA-512 – с 1024-битовыми.

Дополнение блоков

Что, если хешируемое сообщение нельзя представить в виде последовательности полных блоков? Например, если длина блока равна 512 бит, то 520-битовое сообщение будет содержать один блок или еще 8 бит. В таком случае последний блок строится следующим образом: взять оставшийся кусок (в нашем примере 8 бит), дописать в конец бит 1, затем дописать нулевые биты и, наконец, дописать длину исходного сообщения, кодируемую фиксированным числом битов. Такое дополнение гарантирует, что любые два различных сообщения приводят к разным последовательностям блоков, а потому имеют разные хеш-значения.

Например, если 8-битовая строка 10101010 хешируется функцией SHA-256, работающей с 512-битовыми блоками, то первый и единственный блок будет выглядеть следующим образом:

10101010*1000000000000000* ( ) *000000000000<u>1000</u>*

Здесь первые восемь бит (10101010) – это биты сообщения, а все остальные – дополнение (набраны курсивом). Биты *1000* в конце блока (подчеркнуты) кодируют длину сообщения (8 в двоичной записи). Таким образом, в результате дополнения получилось 512-битовое со-

общение, содержащее один 512-битовый блок, который можно обработать функцией сжатия SHA-256.

Гарантии безопасности

Построение Меркла–Дамгора по существу преобразует безопасную функцию сжатия, принимающую небольшие блоки фиксированной длины, в безопасную функцию хеширования, принимающую данные произвольной длины. Если функция сжатия стойкая к коллизиям и восстановлению прообраза, то надстроенная над ней функция хеширования тоже будет обладать этими свойствами. Это верно, потому что любую успешную атаку с восстановлением прообраза для хеш-функции М-Д можно было бы превратить в успешную атаку с восстановлением прообраза функции сжатия, как показали Меркл и Дамгор в статьях 1989 года (см. раздел «Для дополнительного чтения» ниже). То же самое справедливо и для коллизий: противник не сможет преодолеть стойкость хеш-функции к коллизиям, не преодолев стойкости лежащей в ее основе функции сжатия, поэтому безопасность последней гарантирует безопасность хеш-функции.

Отметим, что обратное неверно, поскольку наличие коллизии у функции сжатия необязательно означает коллизию хеш-функции. Коллизия **Compress**(X, M_1) = **Compress**(Y, M_2) для цепных значений X и Y, отличающихся от H_0, не приводит к коллизии хеш-функции, потому что невозможно включить эту коллизию в итеративную цепочку хешей – если только одним из цепных значений не окажется X, а другим Y, но это крайне маловероятно.

Нахождение мультиколлизий

Мультиколлизия случается, когда три или более сообщений хешируются в одно и то же значение. Например, тройка (X, Y, Z) такая, что **Hash**(X) = **Hash**(Y) = **Hash**(Z), называется *3-коллизией*. В идеале нахождение мультиколлизий должно быть гораздо более трудным, чем нахождение коллизий, но существует простой трюк, позволяющий найти их почти с такими же затратами, как одиночную коллизию. Вот как он работает.

1. Найти первую коллизию: **Compress**(H_0, $M_{1.1}$) = **Compress**(H_0, $M_{1.2}$) = H_1. Сейчас мы имеем 2-коллизию, т. е. два сообщения с одинаковым хеш-значением.
2. Найти вторую коллизию, для которой H_1 – начальное цепное значение: **Compress**(H_1, $M_{2.1}$) = **Compress**(H_1, $M_{2.2}$) = H_2. Теперь мы имеем 4-коллизию – четыре сообщения с одинаковым хеш-значением H_2: $M_{1.1}||M_{2.1}$, $M_{1.1}||M_{2.2}$, $M_{1.2}||M_{2.1}$ и $M_{1.2}||M_{2.2}$.
3. Повторив этот процесс N раз, мы найдем 2^N N-блочных сообщений с одинаковым хеш-значением, или 2^N-коллизию, произведя «всего» примерно $N2^N$ вычислений хеш-функции.

На практике этот прием не дает серьезных преимуществ, потому что в любом случае требует нахождения опорной 2-коллизии.

Создание функций сжатия: построение Дэвиса–Мейера

Все функции сжатия, используемые в реальных функциях хеширования, таких как SHA-256 и BLAKE2, основаны на блочных шифрах, потому что это самый простой способ. На рис. 6.5 показана наиболее распространенная схема функций сжатия на основе блочного шифра – *построение Дэвиса–Мейера*.

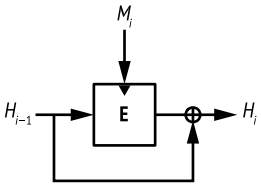

Рис. 6.5. Построение Дэвиса–Мейера. Черный треугольник показывает, откуда поступает ключ блочного шифра

Получив блок сообщения M_i и предыдущее цепное значение H_{i-1}, функция сжатия Дэвиса–Мейера применяет блочный шифр **E** для вычисления нового цепного значения:

$$H_i = \mathbf{E}(M_i, H_{i-1}) \oplus H_{i-1}.$$

Блок сообщения M_i играет роль ключа блочного шифра, а цепное значение H_{i-1} – роль блока открытого текста. При условии что блочный шифр безопасен, получающаяся функция сжатия является безопасной и стойкой к коллизиям и восстановлению прообраза. Без операции XOR с предыдущим цепным значением ($\oplus H_{i-1}$) построение Дэвиса–Мейера было бы небезопасным, потому что его можно было бы обратить, перейдя от нового цепного значения к предыдущему с помощью функции дешифрирования блочного шифра.

Примечание *Построение Дэвиса–Мейера обладает удивительным свойством: для него можно найти неподвижные точки, т. е. цепные значения, которые остаются неизменными после применения функции сжатия с заданным блоком сообщения. Достаточно взять в качестве цепного значения $H_{i-1} = \mathbf{D}(M_i, 0)$, где* **D** *– функция дешифрирования, соответствующая* **E**. *Тогда новое цепное значение H_i будет равно предыдущему, H_{i-1}:*

$$\begin{aligned} H_i = \mathbf{E}(M_i, H_{i-1}) \oplus H_{i-1} &= \mathbf{E}(M_i, \mathbf{D}(M_i, 0)) \oplus \mathbf{D}(M_i, 0) \\ &= 0 \oplus \mathbf{D}(M_i, 0) = \mathbf{D}(M_i, 0) = H_{i-1}. \end{aligned}$$

Мы получаем $H_i = H_{i-1}$, потому что подстановка результата дешифрирования нуля в функцию шифрования дает нуль – член $\mathbf{E}(M_i, \mathbf{D}(M_i, 0))$ – и в выражении для выхода функции сжатия остается только член $\oplus H_{i-1}$. Таким образом, можно найти неподвижные точки функций сжатия для хеш-функций семейства SHA-2, равно как и для стандартов MD5 и SHA-1, тоже основанных на построении Дэвиса–Мейера. По счастью, наличие неподвижных точек не ставит безопасность под угрозу.

Существует много других функций сжатия на основе блочных шифров, помимо построения Дэвиса–Мейера, например показанные на рис. 6.6, но они не столь популярны, потому что сложнее или требуют, чтобы блок сообщения был такой же длины, как цепное значение.

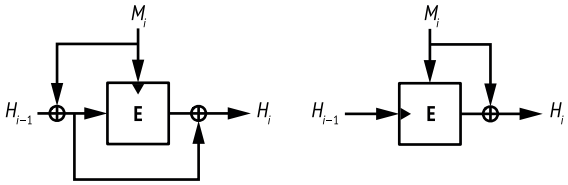

Рис. 6.6. Другие построения безопасных функций сжатия на основе блочных шифров

Хеш-функции на основе перестановок: функции губки

После десятилетий исследований криптографы теперь знают все о методах хеширования на основе блочных шифров. Но все-таки нет ли более простых способов хеширования? Зачем нам блочный шифр – алгоритм, принимающий секретный ключ, – если функциям хеширования секретный ключ не нужен? Почему бы не построить хеш-функцию на основе алгоритма блочного шифра с фиксированным ключом, состоящего из одной перестановки?

Такие упрощенные функции хеширования называются функциями губки, в них используется одна перестановка вместо функции сжатия и блочного шифра (см. рис. 6.7). Для перемешивания битов сообщения с внутренним состоянием в функциях губки применяется не блочный шифр, а операция XOR. Функции губки не только проще функций Меркла–Дамгора, но и более универсальны. Они встречаются и в функциях хеширования, и в качестве детерминированных генераторов случайных битов, потоковых шифров, псевдослучайных функций (см. главу 7) и шифров с аутентификацией (см. главу 8). Самая знаменитая функция губки, Keccak, известна также под названием SHA-3.

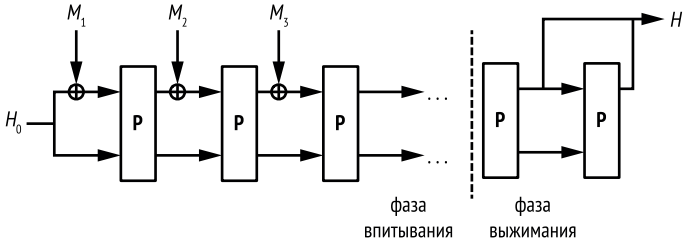

Рис. 6.7. Построение губки

Функция губки работает следующим образом.

1. К первому блоку сообщения M_1 и к предопределенному начальному внутреннему состоянию (например, строке, состоящей из

одних нулей) H_0 применяется операция XOR. Все блоки сообщения одинакового размера, меньшего, чем размер внутреннего состояния.

2. Перестановка **P** преобразует внутреннее состояние в другое значение того же размера.
3. Оно объединяется с блоком M_2 операцией XOR, после чего снова применяется **P**. Это повторяется для блоков M_3, M_4 и т. д. Процесс называется *фазой впитывания*.
4. После обработки всех блоков сообщения губка применяет перестановку **P** еще раз и выделяет из состояния блок битов, образующих хеш. (Если нужен более длинный хеш, то нужно применить **P** еще раз и выделить блок.) Это называется *фазой выжимания*.

Безопасность функции губки зависит от длины ее внутреннего состояния и длины блоков. Если блоки сообщения r-битовые, а внутреннее состояние w-битовое, то существует $c = w - r$ бит внутреннего состояния, которые невозможно модифицировать с помощью блоков сообщения. Значение c называется *емкостью* губки, а уровень безопасности, гарантируемый функцией губки, равен $c/2$. Например, чтобы добиться 256-битовой безопасности при 64-битовых блоках сообщения, внутреннее состояние должно иметь длину $w = 2 \times 256 + 64 = 576$ бит. Разумеется, уровень безопасности зависит также от длины n хеш-значения. Поэтому сложность атаки нахождения коллизий равна минимуму из чисел $2^{n/2}$ и $2^{c/2}$, а сложность атаки восстановления второго прообраза – минимуму из 2^n и $2^{c/2}$.

Чтобы считаться безопасной, перестановка **P** должна вести себя как случайная, т. е. не быть статистически смещенной и не иметь математической структуры, которая позволила бы противнику предсказывать выходы. Как и хеш-функции на основе функций сжатия, функции губки нуждаются в дополнении сообщений, но дополнение в этом случае проще, потому что длину сообщения включать не нужно. За последним битом сообщения просто дописывается бит 1, а затем необходимое количество нулей.

Семейство хеш-функций SHA

Функции хеширования семейства Secure Hash Algorithm (SHA) определены стандартом NIST для применения невоенными федеральными агентствами в США. Он считается международным стандартом, и лишь некоторые страны применяют собственные алгоритмы хеширования (например, SM3 в Китае, Streebog в России и Kupyna на Украине) по причинам, связанным с суверенитетом, а не отсутствием доверия к безопасности SHA. Американский стандарт SHA более скрупулезно изучен криптоаналитиками, чем другие стандарты.

Примечание *Функция хеширования Message Digest 5 (MD5) была самой популярной с 1992 года и до ее взлома примерно в 2005 году, после чего многие приложения переключились на одну из функций семейства SHA. MD5 обрабатывает 512-битовые блоки сообщения и обновляет 128-битовое внутреннее состояние, порождая 128-битовый хеш, т. е. обеспечивает не более 128-битовой стойкости к восстановлению прообраза и 64-битовой стойкости к коллизиям. В 1996 году криптоаналитики предупредили о коллизии функции сжатия MD5, но это предупреждение не имело последствий до 2005 года, когда группа китайских криптоаналитиков описала, как вычислить коллизии для полного MD5-хеша. На момент написания этой книги нахождение коллизии в MD5 занимает всего несколько секунд, и тем не менее многие системы все еще поддерживают MD5, чаще всего ради обратной совместимости.*

SHA-1

Стандарт SHA-1 появился в результате обнаружения ошибки в оригинальной хеш-функции SHA-0, применявшейся АНБ. В 1993 году NIST стандартизовал алгоритм хеширования SHA-0, но в 1995 году АНБ обнародовало функцию SHA-1, исправляющую неназванную проблему безопасности в SHA-0. Причина выяснилась, когда в 1998 году два исследователя обнаружили, как найти коллизии для SHA-0, выполнив примерно 2^{60} операций вместо 2^{80}, ожидаемых для 160-битовых хеш-функций, каковыми являются SHA-0 и SHA-1. Впоследствии сложность атаки удалось уменьшить до примерно 2^{33} операций, что позволяет найти коллизии для SHA-0 меньше, чем за час.

Внутреннее устройство SHA-1

SHA-1 сочетает хеш-функцию Меркла–Дамгора с функцией сжатия Дэвиса–Мейера, основанной на специально сконструированном блочном шифре, который иногда называют SHACAL. То есть SHA-1 повторяет в цикле следующую операцию над 512-битовыми блоками сообщений (M):

$$H = \mathbf{E}(M, H) + H.$$

Здесь употребление знака + вместо $\oplus$ (XOR) – не опечатка. $\mathbf{E}(M, H)$ и H рассматриваются как массивы 32-битовых целых чисел, и слова в соответственных позициях складываются: первое 32-битовое слово $\mathbf{E}(M, H)$ с первым 32-битовым словом H и т. д. Начальное значение H постоянно для любого сообщения, затем H модифицируется согласно приведенной выше формуле, а конечное значение H после обработки всех блоков возвращается в качестве хеш-значения сообщения.

После выполнения блочного шифра с блоком сообщения в качестве ключа и текущим 160-битовым цепным значением в качестве блока открытого текста 160-битовый результат рассматривается как массив пяти 32-битовых слов, каждое из которых складывается с соответственной 32-битовой частью начального значения H.

В листинге 6.4 показана функция сжатия SHA-1, `SHA1-compress()`:

Листинг 6.4. Функция сжатия в SHA-1

```
SHA1-compress(H, M) {
    (a0, b0, c0, d0, e0) = H // разложение H на пять 32-битовых слов
                             // с обратным порядком байтов
    (a, b, c, d, e) = SHA1-blockcipher(a0, b0, c0, d0, e0, M)
    return (a + a0, b + b0, c + c0, d + d0, e + e0)
}
```

Функция блочного шифра SHA-1, `SHA1-blockcipher()`, выделенная полужирным шрифтом в листинге 6.5, принимает 512-битовый блок сообщения *M* в качестве ключа и преобразует пять 32-битовых слов (`a`, `b`, `c`, `d`, `e`), 80 раз повторяя в цикле короткую последовательность операций, чтобы заменить слово комбинацией всех пяти слов. Затем она сдвигает другие слова в массиве, как в регистре сдвига.

Листинг 6.5. Блочный шифр в SHA-1

```
SHA1-blockcipher(a, b, c, d, e, M) {
    W = expand(M)
    for i = 0 to 79 {
        new = (a <<< 5) + f(i, b, c, d) + e + K[i] + W[i]
        (a, b, c, d, e) = (new, a, b >>> 2, c, d)
    }
    return (a, b, c, d, e)
}
```

Функция `expand()`, показанная в листинге 6.6, создает массив восьмидесяти 32-битовых слов, *W*, из блока сообщения длиной 16 слов. Для этого в первые 16 слов *W* записываются слова *M*, а в последующие – XOR-комбинации предыдущих слов, циклически сдвинутые на один бит влево.

Листинг 6.6. Функция `expand()` *в SHA-1*

```
expand(M) {
    // 512-битовый блок M рассматривается как массив шестнадцати 32-битовых слов
    W = пустой массив из восьмидесяти 32-битовых слов
    for i = 0 to 79 {
        if i < 16 then W[i] = M[i]
        else
            W[i] = (W[i - 3] ⊕ W[i - 8] ⊕ W[i - 14] ⊕ W[i - 16]) <<< 1
    }
    return W
}
```

Операция <<< 1 в листинге 6.6 – единственное различие между функциями SHA-1 и SHA-0.

Наконец, функция `f()` (см. листинг 6.7) в `SHA1-blockcipher()` выполняет последовательность простых поразрядных логических операций (булевых функций), зависящую от номера раунда.

Листинг 6.7. Функция `f()` *в SHA-1*

```
f(i, b, c, d) {
    if i < 20 then return ((b & c) ⊕ (~b & d))
    if i < 40 then return (b ⊕ c ⊕ d)
    if i < 60 then return ((b & c) ⊕ (b & d) ⊕ (c & d))
    if i < 80 then return (b ⊕ c ⊕ d)
}
```

Вторая и четвертая булевы функции в листинге 6.7 – просто XOR трех входных слов, это линейная операция. Напротив, первая и третья функции включают нелинейный оператор `&` (логическое И), чтобы защититься от дифференциального криптоанализа, который, как вы помните, эксплуатирует предсказуемое распространение различий в битах. Без оператора `&` (иными словами, если бы `f()` всегда вычисляла, к примеру, `b` ⊕ `c` ⊕ `d`) SHA-1 было бы легко взломать, проследив закономерности во внутреннем состоянии.

Атаки на SHA-1

Хотя функция SHA-1 безопаснее, чем SHA-0, она все равно небезопасна, поэтому браузер Chrome отмечает сайты, использующие SHA-1 при установлении HTTPS-соединения как небезопасные. Хотя 160-битовый хеш должен обеспечивать 80-битовую стойкость к коллизиям, в 2005 году в SHA-1 были обнаружены слабости, и, по некоторым оценкам, для нахождения коллизии должно хватить приблизительно 2^{63} вычислений (если бы алгоритм не содержал дефектов, то это число равнялось бы 2^{80}). Реальная коллизия в SHA-1 была найдена лишь спустя 12 лет, когда после нескольких лет криптоанализа Марк Стивенс и другие исследователи представили два PDF-документа с одинаковым хеш-значением (см. *https://shattered.io/*).

Вывод: использовать SHA-1 не следует. Как уже было отмечено, браузеры помечают функцию SHA-1 как небезопасную, и NIST ее больше не рекомендует. Используйте вместо нее функции хеширования семейства SHA-2, BLAKE2 или SHA-3.

SHA-2

Алгоритм SHA-2, пришедший на смену SHA-1, был спроектирован АНБ и стандартизован NIST. SHA-2 – это семейство из четырех функций хеширования: SHA-224, SHA-256, SHA-384 и SHA-512, из которых основными являются SHA-256 и SHA-512. Трехзначное число обозначает длину хеш-значения в битах.

SHA-256

Побудительным мотивом для разработки SHA стало желание генерировать более длинные хеши и тем самым обеспечить более высокий уровень безопасности, чем SHA-1. Например, если в SHA-1 цепные значения 160-битовые, то в SHA-256 они занимают 256 бит, т. е. восемь 32-битовых слов. И SHA-1, и SHA-256 работают с 512-битовыми блоками сообщений, но если SHA-1 выполняет 80 раундов, то SHA-256 – 64 раунда, расширяя 16-словный блок сообщения в 64-словный с помощью функции `expand256()`, показанной в листинге 6.8.

Листинг 6.8. Функция `expand256()` *из SHA-256*

```
expand256(M) {
    // 512-битовый блок M рассматривается как массив шестнадцати 32-битовых слов
    W = пустой массив, содержащий 64 32-битовых слова
    for i = 0 to 63 {
        if i < 16 then W[i] = M[i]
        else {
            // the ">>" - сдвиг вправо, тогда как ">>>" - циклический сдвиг
            // вправо, это не опечатка
            s0 = (W[i - 15] >>> 7) ⊕ (W[i - 15] >>> 18) ⊕ (W[i - 15] >> 3)
            s1 = (W[i - 2] >>> 17) ⊕ (W[i - 2] >>> 19) ⊕ (W[i - 2] >> 10)
            W[i] = W[i - 16] + s0 + W[i - 7] + s1
        }
    }
    return W
}
```

Заметим, что функция расширения сообщения `expand256()` в SHA-2 сложнее, чем функция `expand()` в SHA-1 (см. листинг 6.6), которая выполняет только XOR и циклический сдвиг на 1 бит. Главный цикл функции сжатия в SHA-256 тоже сложнее, чем в SHA-1, в нем выполняется 26 арифметических операций, а не 11. Операциями являются XOR, логическое AND и циклический сдвиг слова.

Другие алгоритмы семейства SHA-2

Функция SHA-224 из семейства SHA-2 алгоритмически идентична SHA-256, только в качестве начального значения берется другое множество восьми 32-битовых слов, а длина хеш-значения составляет не 256 бит, а 224 – первые 224 бита последнего цепного значения.

Семейство SHA-2 включает также алгоритмы SHA-512 и SHA-384. SHA-512 похож на SHA-256, но работает с 64-битовыми, а не 32-битовыми словами. Поэтому используются 512-битовые цепные значения (восемь 64-битовых слов), блоки сообщения имеют длину 1024 бита (шестнадцать 64-битовых слов), а количество раундов равно 80, а не 64. Функция сжатия почти такая же, как в SHA-256, хотя из-за более широких слов величины циклических сдвигов другие. (Например, SHA-512 включает операцию `a >>> 34`, которая не имеет смысла для

32-битовых слов в SHA-256.) SHA-384 по отношению к SHA-512 занимает такое же место, как SHA-224 по отношению к SHA-256, – это тот же алгоритм, но с другим начальным значением, а конечное хеш-значение усекается до 384 бит.

С точки зрения безопасности, все четыре варианта SHA-2 до сих пор выполняют обещания: SHA-256 гарантирует 256-битовую стойкость к восстановлению прообраза, SHA-512 гарантирует примерно 256-битовую стойкость к коллизиям и т. д. Тем не менее не существует формального доказательства безопасности функций из семейства SHA-2, мы говорим лишь о предположительной безопасности.

Впрочем, после практических атак на MD5 и SHA-1 исследователи, а вместе с ними NIST стали проявлять обеспокоенность по поводу долговременной безопасности SHA-2, поскольку эти функции похожи на SHA-1, и многие полагают, что успешные атаки на SHA-2 – просто вопрос времени. Но на момент написания этой книги таковых не наблюдалось. Тем не менее NIST разработал резервный план: SHA-3.

Конкурс на звание SHA-3

Объявленный в 2007 году конкурс хеш-функций NIST (официальное название конкурса на звание SHA-3) начался с призыва присылать предложения и некоторых базовых требований: предлагаемые хеш-функции должны быть не менее безопасными и быстрыми, чем SHA-2, и должны уметь делать, по крайней мере, то же, что умеет SHA-2. Кроме того, кандидаты не должны быть слишком похожи на SHA-1 и SHA-2, чтобы противостоять атакам, способным взломать SHA-1 и потенциально SHA-2. К 2008 году NIST получил 64 предложения со всего мира, в т. ч. от университетов и крупных корпораций (в частности, BT, IBM, Microsoft, Qualcomm и Sony). Из них 51 предложение отвечало требованиям и приняло участие в первом туре конкурса.

На протяжении первых недель конкурса криптоаналитики безжалостно атаковали предложения. В июле 2009 года NIST объявил 14 кандидатов, прошедших во второй тур. Спустя 15 месяцев, в течение которых эти кандидаты подвергались анализу и оценке, NIST отобрал 5 финалистов.

- **BLAKE**. Усовершенствованная хеш-функция Меркла–Дамгора, в которой функция сжатия основана на блочном шифре, который, в свою очередь, основан на базовой функции потокового шифра ChaCha, содержащей цепочку операций сложения, XOR и циклического сдвига слов. BLAKE была спроектирована командой ученых из университетов Швейцарии и Великобритании, в которую входил и я.
- **Grøstl.** Усовершенствованная хеш-функция Меркла–Дамгора, в которой функция сжатия включает две перестановки (или блочные шифры с фиксированным ключом), основанные на базовой

функции блочного шифра AES. Grøstl был спроектирован группой семи исследователей из университетов Дании и Австрии.

- **JH.** Модифицированное построение функции губки, в которой блоки сообщения подаются до и после перестановки, а не только до нее. Перестановка выполняет также операции, похожие на подстановочно-перестановочный блочный шифр (см. главу 4). JH была спроектирована криптографом из Сингапурского университета.
- **Keccak.** Функция губки, в которой перестановка выполняет только поразрядные операции. Keccak была спроектирована группой четырех криптографов из компании по производству полупроводниковых изделий, базирующейся в Бельгии и Италии и включавшей одного из двух авторов AES.
- **Skein.** Хеш-функция, основанная на ином режиме работы, чем построение Меркла–Дамгора. Вместе с тем функция сжатия основана на новаторском блочном шифре, в котором используются только операции целочисленного сложения, XOR и циклического сдвига слов. Skein была спроектирована группой восьми криптографов из академических учреждений и промышленных компаний из США, включая знаменитого Брюса Шнейера.

После скрупулезного анализа пяти финалистов NIST объявил победителя: Keccak. В отчете NIST отмечается «элегантный дизайн, большой запас безопасности, высокая общая производительность, отличная эффективность при аппаратной реализации и гибкость». Посмотрим, как работает Keccak.

Keccak (SHA-3)

Одна из основных причин, по которой NIST остановился на Keccak, заключалась в том, что она абсолютно не похожа на SHA-1 и SHA-2. Прежде всего это функция губки. Базовым алгоритмом Keccak является перестановка 1600-битового состояния, которой подаются блоки размером 1152, 1088, 832 или 576 бит, в результате чего порождаются хеш-значения длиной 224, 256, 384 или 512 бит соответственно – те же четыре длины, что у хеш-функций из семейства SHA-2. Однако в SHA-3 используется один базовый алгоритм для всех четырех длин хеша, а не два, как в SHA-2.

Еще одна причина выбора Keccak – то, что это больше, чем функция хеширования. В документе FIPS 202, стандартизующем SHA-3, определены четыре хеш-функции: SHA3-224, SHA3-256, SHA3-384 и SHA3-512 – и два алгоритма: SHAKE128 и SHAKE256. (*SHAKE* – акроним «Secure Hash Algorithm with Keccak».) Эти два алгоритма являются *функциями с расширяемым выходом* (extendable-output function – XOF), т. е. хеш-функциями, способными порождать хеши переменной длины, даже очень большой. Числа 128 и 256 обозначают уровень безопасности каждого алгоритма.

Сам по себе стандарт FIPS 202 весьма длинный и читается с трудом, но можно найти реализации с открытым исходным кодом, которые работают довольно быстро, – понять алгоритм, глядя на код, проще, чем читая спецификации. Например, реализация tiny_sha3 (*https://github.com/mjosaarinen/tiny_sha3/*), распространяемая по лицензии MIT и написанная Маркку-Юхани О. Саариненом, содержит объяснение базового алгоритма Keccak в виде 19 строк кода на C, частично воспроизведенных в листинге 6.9.

Листинг 6.9. Реализация tiny_sha3

```
static void sha3_keccakf(uint64_t st[25], int rounds)
{
    (⊕)
    for (r = 0; r < rounds; r++) {

      ❶ // Theta
        for (i = 0; i < 5; i++)
            bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];

        for (i = 0; i < 5; i++) {
            t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
            for (j = 0; j < 25; j += 5)
                st[j + i] ^= t;
        }

      ❷ // Rho Pi
        t = st[1];
        for (i = 0; i < 24; i++) {
            j = keccakf_piln[i];
            bc[0] = st[j];
            st[j] = ROTL64(t, keccakf_rotc[i]);
            t = bc[0];
        }

      ❸ // Chi
        for (j = 0; j < 25; j += 5) {
            for (i = 0; i < 5; i++)
                bc[i] = st[j + i];
            for (i = 0; i < 5; i++)
                st[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
        }

      ❹ // Iota
        st[0] ^= keccakf_rndc[r];
    }
    (⊕)
}
```

Программа tiny_sha3 реализует перестановку **P** алгоритма Keccak, обратимое преобразование 1600-битового состояния, рассматриваемого как массив двадцати пяти 64-битовых слов. Видно, что в цикле

выполняется серия раундов, каждый из которых состоит из четырех основных шагов (обозначенных ❶, ❷, ❸ и ❹).

- Первый шаг, `Theta` ❶, включает операции XOR между 64-битовыми словами или результатами циклического сдвига слов на 1 бит (операция `ROTL64(w, 1)` циклически сдвигает слово `w` на 1 бит).
- Второй шаг, `Rho Pi` ❷, включает циклические сдвиги 64-битовых слов на константы, хранящиеся в массиве `keccakf_rotc[]`.
- Третий шаг, `Chi` ❸, включает дополнительные операции XOR, а также логические операции AND (оператор `&`) между 64-битовыми словами. Эти AND – единственные нелинейные операции в Keccak, именно они придают ему криптографическую стойкость.
- Четвертый шаг, `Iota` ❹, включает операцию XOR с 64-битовой константой, хранящейся в массиве `keccakf_rndc[]`.

Эти операции придают стойкость алгоритму перестановки в SHA-3, избавляя его от статистического смещения и допускающей эксплуатацию структуры. SHA-3 – плод более десяти лет исследований, и сотни квалифицированных криптографов так и не смогли взломать его. Маловероятно, что он будет взломан в обозримом будущем.

Функция хеширования BLAKE 2

Безопасность, конечно, стоит на первом месте, но на втором находится быстродействие. Я встречал много случаев, когда разработчик отказывался переходить с MD5 на SHA-1 просто потому, что MD5 быстрее, или с SHA-1 на SHA-2, потому что SHA-2 работает заметно медленнее SHA-1. К сожалению, SHA-3 не быстрее SHA-2, а поскольку SHA-2 все еще считается безопасным, найдется мало стимулов для перехода на SHA-3. А есть ли алгоритм, который работает быстрее SHA-1 и SHA-2 и при этом даже более безопасен? Ответ дает функция хеширования BLAKE2, опубликованная уже после конкурса на звание SHA-3.

Полное раскрытие информации: проектировал BLAKE2 я сам вместе с Сэмюэлем Нивсом, Руко Уилкоксом О'Хирном и Кристианом Виннерлейном.

При проектировании BLAKE2 ставились следующие цели:

- она должна быть не менее безопасной, чем SHA-3, а желательно даже более криптостойкой;
- она должна работать быстрее всем предыдущих стандартов хеширования, включая MD5;
- она должна быть пригодна для использования в современных приложениях и уметь хешировать большие объемы данных, представленные как в виде нескольких больших сообщений, так и в виде множества мелких сообщений, с секретным ключом или без него;

- она должна хорошо работать на современных процессорах, поддерживающих параллельные вычисления на нескольких ядрах, а также параллелизм на уровне команд на одном ядре.

Итогом проектирования стали две функции хеширования:

- BLAKE2b (или просто BLAKE2), оптимизированная для 64-разрядных платформ, порождает дайджесты длиной от 1 до 64 байт;
- BLAKE2s, оптимизированная для 8–32-разрядных платформ, может порождать дайджесты длиной от 1 до 32 байт.

У каждой функции есть параллельный вариант, способный задействовать несколько процессорных ядер. Параллельная версия BLAKE2b, BLAKE2bp, работает на четырех ядрах, а BLAKE2sp – на восьми ядрах. Первая функция работает быстрее всех на современных серверах и ноутбуках, на ноутбуке ее быстродействие приближается к 2 Гб/с. На самом деле BLAKE2 – самая быстрая из имеющихся на сегодняшний день хеш-функций; благодаря ее скорости и функциональности она стала самой популярной среди функций, не стандартизованных NIST. BLAKE2 используется в бесчисленных приложениях и включена во все важные криптографические библиотеки, в т. ч. OpenSSL и Sodium.

Примечание *Спецификации и эталонный код BLAKE2 находятся по адресу https://blake2.net/, а оптимизированный код и библиотеки можно скачать по адресу https://github.com/BLAKE2/. В эталонном коде имеется также расширение BLAKE2X, умеющее порождать хеш-значения произвольной длины.*

Функция сжатия BLAKE2, показанная на рис. 6.8, – это вариант построения Дэвиса–Мейера, который принимает дополнительные параметры – *счетчик* (гарантирующий, что все функции сжатия ведут себя по-разному) и *флаг* (показывающий, обрабатывает ли функция сжатия последний блок сообщения, – это повышает безопасность).

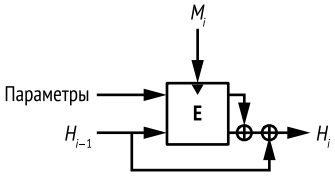

Рис. 6.8. Функция сжатия BLAKE2. Обе половины состояния объединяются с помощью XOR после применения блочного шифра

Блочный шифр в функции сжатия BLAKE2 основан на потоковом шифре ChaCha, который сам является вариантом потокового шифра Salsa20, рассмотренного в главе 5. Внутри блочного шифра базовая операция BLAKE2b состоит из следующей цепочки операций, которая преобразует четыре 64-битовых слова состояния с помощью двух слов сообщения M_i и M_j:

$a = a + b + M_i$;
$d = ((d \oplus a) >>> 32)$;
$c = c + d$;
$b = ((b \oplus c) >>> 24)$;
$a = a + b + M_j$;
$d = ((d \oplus a) >>> 16)$;
$c = c + d$;
$b = ((b \oplus c) >>> 63)$.

Базовая операция BLAKE2s похожа, но работает с 32-битовыми, а не 64-битовыми словами (поэтому величины циклических сдвигов другие).

Какие возможны проблемы

Несмотря на кажущуюся простоту, функции хеширования могут стать причиной серьезных проблем безопасности, если используются неправильно или не в том месте, например когда для проверки целостности файла в приложениях, передающих данные по сети, вместо криптографической хеш-функции применяется нестойкий алгоритм вычисления контрольной суммы типа CRC. Однако эта слабость бледнеет по сравнению с другими, способными привести к полной компрометации на первый взгляд безопасных хеш-функций. Мы рассмотрим два примера таких катастроф: первый относится к SHA-1 и SHA-2, но не к BLAKE2 или SHA-3, второй – ко всем четырем функциям.

Атака удлинением сообщения

Атака удлинением сообщения, показанная на рис. 6.9, – главная угроза построению Меркла–Дамгора.

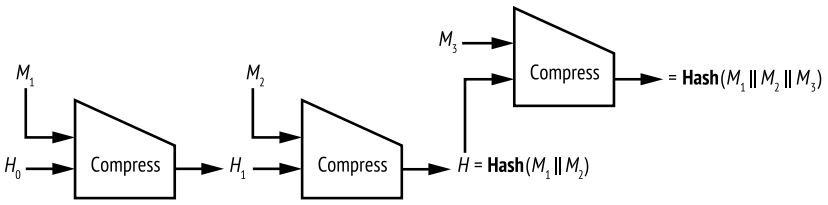

Рис. 6.9. Атака удлинением сообщения

Суть ее в следующем: если мы знаем **Hash**(M) для некоторого *неизвестного* сообщения M, состоящего из блоков M_1 и M_2 (после дополнения), то можем определить **Hash**($M_1 \| M_2 \| M_3$) для любого блока M_3. Поскольку хеш $M_1 \| M_2$ – это цепное значение, следующее непосредственно за M_2, можно добавить еще один блок M_3 к хешированному сообщению, даже не зная, какие данные хешировались. Более того,

этот прием обобщается на любое число блоков в неизвестном сообщении (здесь ($M_1 || M_2$) или в суффиксе (здесь M_5).

Атака удлинением сообщения неприменима к большинству приложений функций хеширования, но может скомпрометировать безопасность, если хеш используется чересчур творчески. К сожалению, хеш-функции семейства SHA-2 уязвимы к атаке удлинением сообщения, несмотря на то что проектировавшее их АНБ и отвечавший за стандартизацию NIST прекрасно знали об этом дефекте. Данного дефекта можно было бы избежать, просто сделав последний вызов функции сжатия отличающимся от всех остальных (например, добавив параметр, который принимает значение 1, тогда как для предыдущих вызовов он равен 0). Именно так, кстати, и поступает BLAKE2.

Обман протоколов доказательства хранения

В облачных приложениях хеш-функции используются в протоколах *доказательства хранения*, с помощью которых сервер (поставщик облачных услуг) доказывает клиенту (пользователю услуг облачного хранения), что он действительно хранит файлы, который должен хранить по поручению клиента.

В 2007 году в статье Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin «SafeStore: A Durable and Practical Storage System» (*https://www.cs.utexas.edu/~lorenzo/papers/p129-kotla.pdf*) предложили следующий протокол доказательства хранения, верифицирующий хранение файла *M*.

1. Клиент выбирает случайное значение *C*, *вопрос*.
2. Сервер вычисляет *ответ* **Hash**($M || C$) и отправляет его клиенту.
3. Клиент также вычисляет **Hash**($M || C$) и проверяет, что результат совпадает со значением, полученным от сервера.

Идея заключается в том, что сервер не сможет обмануть клиента, потому что, не зная *M*, не сумеет угадать значение **Hash**($M || C$). Но тут есть подвох: на самом деле функция **Hash** итеративно обрабатывает входные данные поблочно, вычисляя промежуточные цепные значения между блоками. Например, если **Hash** – хеш-функция SHA-256, а *M* – 512-битовый блок (размер блока в SHA-256), то сервер может обмануть клиента. Как? При первом получении *M* он вычисляет H_1 = **Compress**(H_0, M_1), цепное значение, полученное SHA-256 по начальному значению, H_0, 512-битовому *M*. Затем он сохраняет H_1 в памяти и отбрасывает *M*; начиная с этого момента он может уже не хранить *M*.

Теперь, когда клиент посылает случайное значение *C*, сервер вычисляет **Compress**(H_1, *C*), дополнив предварительно *C* до полного блока, и возвращает результат в виде **Hash**($M || C$). Поскольку сервер вернул правильное значение **Hash**($M || C$), клиент верит, что сервер хранит полное сообщение, хотя, как мы только что видели, это может быть не так.

Этот трюк будет работать для SHA-1, SHA-2, а также для SHA-3 и BLAKE2. Решение простое: запрашивать **Hash**($C || M$), а не **Hash**($M || C$).

Для дополнительного чтения

Если вы хотите больше узнать о функциях хеширования, почитайте классические работы 1980-х и 1990-х годов, например Ralph Merkle «One Way Hash Functions and DES» и Ivan Dåmgerd «A Design Principle for Hash Functions». Кроме того, познакомьтесь с первым полным исследованием хеширования на основе блочных шифров: Preneel, Govaerts, and Vandewalle «Hash Functions Based on Block Ciphers: A Synthetic Approach».

Дополнительные сведения о поиске коллизий имеются в статье van Oorschot and Wiener «Parallel Collision Search with Cryptanalytic Applications», вышедшей в 1997 году. Чтобы больше узнать о теоретических основах стойкости к коллизиям и восстановлении прообраза, а также об атаках удлинением сообщения, поищите в сети по ключевому слову *indifferentiability* (неразличимость).

Более поздние исследования по хеш-функциям см. в архивах конкурса на звание SHA-3, где имеются описания всех поданных на конкурс алгоритмов и их взлома. Много ссылок можно найти на сайте SHA-3 Zoo по адресу *http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo* и на странице NIST *http://csrc.nist.gov/groups/ST/hash/sha-3/*.

Дополнительные сведения о победителе конкурса на звание SHA-3, Keccak, и функциях губки см. на официальных страницах авторов Keccak *http://keccak.noekeon.org/* и *http://sponge.noekeon.org/*.

И наконец, поинтересуйтесь следующими двумя примерами эксплуатации слабых хеш-функций:

- компьютерный червь Flame, эксплуатируя коллизию в MD5, создавал поддельный сертификат и притворялся легитимной программой;
- в игровой консоли Xbox для построения хеш-функции использовался слабый блочный шифр TEA. Это было использовано для взлома консоли и выполнения на ней произвольного кода.

7

ХЕШИРОВАНИЕ С СЕКРЕТНЫМ КЛЮЧОМ

Хеш-функции, обсуждавшиеся в главе 6, принимают сообщение и возвращают его хеш-значение – обычно короткую строку длиной 256 или 512 бит. Любой желающий может вычислить хеш-значение сообщения и проверить, что конкретное сообщение хешируется в конкретное значение, потому что в этом процессе нет никакого секретного ключа, но иногда мы не хотим, чтобы эту операцию мог выполнить кто угодно. На подобный случай и нужны функции хеширования с секретным ключом.

Существует два важных класса криптографических алгоритмов, применяемых для хеширования с секретным ключом: *имитовставки* (message authentication code – MAC), которые удостоверяют подлинность сообщения и защищают его целостность, и *псевдослучайные функции* (pseudorandom function – PRF), которые порождают кажущие-

ся случайными значения. В первом разделе этой главы мы поговорим о том, чем и почему алгоритмы MAC и PRF похожи, а затем обсудим, как они работают. Одни MAC и PRF основаны на хеш-функциях, другие – на блочных шифрах, а третьи имеют оригинальный дизайн. Наконец, мы рассмотрим примеры атак на имитовставки.

Имитовставки (MAC)

MAC защищает целостность и подлинность сообщения путем создания значения $T = \mathbf{MAC}(K, M)$, называемого аутентификационным жетоном сообщения, M (часто его-то называют имитовставкой M, что ведет к путанице). Точно так же, как можно дешифрировать сообщение, зная ключ шифра, можно проверить, что сообщение не было модифицировано, если знать ключ MAC.

Например, предположим, что Алекс и Билл разделяют ключ, K, и Алекс посылает Биллу сообщение M вместе с аутентификационным жетоном $T = \mathbf{MAC}(K, M)$. Получив сообщение и его аутентификационный жетон, Билл тоже вычисляет $\mathbf{MAC}(K, M)$ и проверяет, что результат равен полученному жетону. Поскольку вычислить это значение мог только Алекс, Билл может быть уверен, что сообщение не было случайно или злонамеренно повреждено в процессе передачи (подтверждение целостности) и что послал его именно Алекс (подтверждение подлинности).

MAC как часть безопасной системы связи

В безопасных системах связи шифр и MAC часто комбинируются для защиты конфиденциальности, целостности и подлинности сообщения. Например, протоколы Internet Protocol Security (IPSec), Secure Shell (SSH) и Transport Layer Security (TLS) генерируют MAC для каждого передаваемого сетевого пакета.

Не во всех системах связи применяется MAC. Иногда аутентификационный жетон добавляет неприемлемые накладные расходы к каждому пакету, особенно если размер пакета составляет от 64 до 128 бит. Например, в стандартах мобильной телефонии 3G и 4G голосовые пакеты шифруются, но не аутентифицируются. Противник может модифицировать зашифрованный звуковой сигнал, и получатель этого не заметит. Таким образом, если противник повредит зашифрованный голосовой пакет, то результатом дешифрирования будет шум, воспринимаемый как статические помехи.

Атаки с подделкой и подобранным сообщением

Что значит «безопасная MAC»? Прежде всего, как и для шифра, ключ должен храниться в секрете. Если MAC безопасна, то противник не должен иметь возможность создать жетон сообщения, не зная ключа. Такая сгенерированная пара из сообщения и жетона называется *под-*

делкой, а восстановление ключа – просто частный случай более общего класса *атак с подделкой*. Аспект безопасности, постулирующий, что найти подделку невозможно, называется *невозможностью подделки*. Очевидно, что должно быть невозможно восстановить секретный ключ по списку жетонов, иначе противник смог бы подделывать жетоны, пользуясь этим ключом.

Что может сделать противник для взлома MAC? Иначе говоря, какова модель атаки? Самая простая модель – *атака с известным сообщением*, когда противник пассивно собирает сообщения и ассоциированные с ними жетоны (например, прослушивая сеть). Но реальный противник организует более действенные атаки, потому что может выбирать, какие сообщения подлежат аутентификации, а значит, получать MAC интересующих его сообщений. Поэтому стандартная модель – *атака с подобранным сообщением*, когда противник получает жетоны для сообщений по своему выбору.

Атаки повторным воспроизведением

MAC не защищены от атак *повторным воспроизведением* жетонов. Например, если можно организовать прослушивание канала связи между Алексом и Биллом, то можно запомнить сообщение, отправленное Алексом Биллу, и его жетон, а позже отправить их Биллу снова, притворившись Алексом. Чтобы предотвратить такие атаки, протокол включает в каждое сообщение его порядковый номер. Этот номер увеличивается для каждого сообщения и аутентифицируется вместе с сообщением. Принимающая сторона получает сообщения с номерами 1, 2, 3, 4 и т. д. Следовательно, если противник попытается отправить сообщение с номером 1 повторно, то получатель увидит, что сообщение пронумеровано не по порядку, т. е. может быть воспроизведением ранее посланного сообщения с номером 1.

Псевдослучайные функции (PRF)

PRF – это функция, которая использует секретный ключ для возврата кажущегося случайным значения **PRF**(*K*, *M*). Поскольку ключ секретный, противник не может предсказать выходные значения.

В отличие от MAC, PRF применяются не самостоятельно, а в составе криптографического алгоритма или протокола. Например, PRF можно использовать для создания блочных шифров в рамках построения Фейстеля, которое рассматривалось в разделе «Как устроены блочные шифры» главы 4. В схемах формирования ключа PRF используются для генерирования криптографических ключей по главному ключу или паролю, а в схемах идентификации – для генерирования ответа на случайный вопрос (когда сервер посылает случайное сообщение-вопрос *M*, а клиент возвращает в ответ **PRF**(*K*, *M*), чтобы доказать, что знает *K*). В стандарте телефонии 4G PRF используется для аутентификации SIM-карты и обслуживающего ее оператора, и похожая PRF

служит также для генерирования ключа шифрования и ключа MAC, используемого в процессе телефонного вызова. В протоколе TLS псевдослучайная функция генерирует материал для ключей по главному ключу, а также сеансовые случайные значения. PRF встречается даже в некриптографической функции `hash()`, встроенной в язык Python для сравнения объектов.

Безопасность PRF

Чтобы считаться безопасной, псевдослучайная функция не должна содержать никаких закономерностей в выходных значениях, т. е. последние должны быть неотличимы от истинно случайных. Противник, не знающий ключа *K*, не должен иметь возможность отличить результаты **PRF**(*K*, *M*) от случайных значений. Иначе говоря, у противника не должно быть способа узнать, имеет ли он дело с алгоритмом PRF или со случайной функцией. Эрудиты называют это *неотличимостью от случайной функции*. (Дополнительные сведения о теоретических основаниях PRF см. в разделе 3.6 первого тома книги Goldreich «Foundations of Cryptography».)

Почему PRF более стойкие, чем MAC

И PRF, и MAC – хеш-функции с секретным ключом, но PRF принципиально более стойкие в основном потому, что к безопасности MAC предъявляются более слабые требования. Если MAC считается безопасной, коль скоро жетоны невозможно подделать, т. е. нельзя угадать выходное значение MAC, то PRF безопасна, только если ее результат нельзя отличить от случайной строки, а это более сильное требование. Если результаты PRF неотличимы от случайных строк, то эти значения, конечно же, нельзя угадать, иными словами, любая безопасная PRF является также безопасной MAC.

Однако обратное неверно: безопасная MAC необязательно является безопасной PRF. Например, предположим, что мы начали с безопасной PRF, **PRF1**, и хотим построить другую безопасную PRF, **PRF2**, следующим образом:

$$\mathbf{PRF2}(K, M) = \mathbf{PRF1}(K, M) \,||\, 0.$$

Поскольку выход **PRF2** определен как выход **PRF1**, за которым следует один нулевой бит, он не выглядит так же случайно, как истинно случайная строка; отличить результаты **PRF2** можно по этому нулевому биту, и, значит, она не является безопасной. Но поскольку **PRF1** безопасна, то **PRF2** по-прежнему представляет собой безопасную MAC. Почему так? Потому что если бы мы могли подделать жетон $T = \mathbf{PRF2}(K, M)$ для некоторого M, то смогли бы подделать и жетон для **PRF1**, что, как мы знаем, невозможно, т. к. **PRF1** является безопасной MAC. Таким образом, **PRF2** – хеш-функция с секретным ключом, которая является примером безопасной MAC, но не безопасной PRF.

Однако не стоит волноваться: таких построений MAC в реальных приложениях вы не встретите. На самом деле многие реально встречающиеся или стандартизованные MAC являются также безопасными PRF и нередко используются в качестве шифров. Например, в TLS алгоритм HMAC-SHA-256 используется и как MAC, и как PRF.

Создание хешей с секретным ключом по хешам без ключа

На протяжении всей истории криптографии MAC и PRF редко проектировались с нуля, обычно они строятся на основе уже существующих алгоритмов, чаще всего хеш-функций блочных шифров. Один кажущийся очевидным способ построения хеш-функции с секретным ключом – подать на вход хеш-функции без ключа сообщение и ключ, но это проще сказать, чем сделать. Обсудим подробнее.

Построение секретного префикса

Для начала рассмотрим метод *построения секретного префикса*, который превращает обычную хеш-функцию в функцию с секретным ключом, для чего дописывает ключ в начало сообщения и возвращает **Hash**($K||M$). Хотя этот подход не всегда дефектный, он может оказаться небезопасным, если хеш-функция уязвима к атаке удлинением сообщения (см. обсуждение в главе 6) или поддерживает ключи разной длины.

Небезопасность относительно атак удлинением сообщения

Напомним (см. главу 6), что хеш-функции из семейства SHA-2 позволяют противнику вычислить хеш частично неизвестного сообщения, зная хеш его укороченного варианта. Формально атака *удлинением сообщения* позволяет вычислить **Hash**($K||M_1||M_2$), если известен только **Hash**($K||M_1$), а M_1 и K неизвестны. Такие функции дают противнику возможность подделать действительные аутентификационные жетоны, потому что не предполагается, что они в состоянии угадать MAC конкатенации $M_1||M_2$, зная только MAC M_1. Из-за этого построение секретного префикса оказывается таким же небезопасным, как MAC и PRF, при использовании, например, с функциями хеширования SHA-256 или SHA-512. Сама возможность атак удлинением сообщения – слабость построения Меркла–Дамгора, финалисты конкурса на звание SHA-3 ее лишены. Способность противостоять атакам удлинением сообщения была одним из требований к SHA-3.

Небезопасность относительно ключей разной длины

Построение секретного префикса небезопасно также, если допускается использование ключей разной длины. Например, если ключ K – 24-битовая шестнадцатеричная строка 123abc, а M равно def00, то

Hash() будет обрабатывать значение $K||M$ = 123abcdef00. Если же K – 16-битовая строка 123a, а M равно bcdef000, то **Hash**() также будет обрабатывать $K||M$ = 123abcdef00. Следовательно, результат построения секретного префикса **Hash**($K||M$) будет одинаков для обоих ключей.

Эта проблема не зависит от хеш-функции, а для ее исправления достаточно хешировать длину ключа вместе с ключом и сообщением, например закодировав длину ключа в битах 16-битовым целым числом L и вычислив хеш **Hash**($L||K||M$). Но делать это необязательно. Современные хеш-функции, в частности BLAKE2 и SHA-3, включают режим, который обходит эти подводные камни и дает безопасную PRF и безопасную MAC.

Построение секретного суффикса

Вместо того чтобы помещать ключ перед сообщением, а затем хешировать результат, как при построении секретного префикса, мы можем поместить его *после*. Именно так работает процедура построения секретного суффикса: строит PRF в виде **Hash**($M||K$).

Дописывание ключа в конец меняет всё. Прежде всего атака удлинением сообщения, работающая против построения MAC с секретным префиксом, бессильна против секретного суффикса. Применив удлинение к MAC с секретным суффиксом, мы получим **Hash**($M_1||K||M_2$) по **Hash**($M_1||K$), но такая атака ничего не дает, потому что **Hash**($M_1||K||M_2$) не является действительной MAC с секретным суффиксом – ключ должен находиться в конце.

Однако построение с секретным суффиксом уязвимо для атаки другого типа. Предположим, что имеется коллизия **Hash**(M_1) = **Hash**(M_2), где M_1 и M_2 – разные сообщения, возможно, разной длины. В случае хеш-функции типа SHA-256 отсюда следует, что **Hash**($M_1||K$) и **Hash**($M_2||K$) тоже совпадают, потому что внутри ключ K будет обрабатываться на основе ранее хешированных данных, а именно **Hash**(M_1) и равного ему **Hash**(M_2). Поэтому мы получим одинаковое хеш-значение вне зависимости от того, находится ключ K после M_1 или после M_2, и значение самого ключа K при этом не играет роли.

Для эксплуатации этого свойства противник должен:

1) найти два сообщения, M_1 и M_2, образующих коллизию;
2) запросить MAC-жетон **Hash**($M_1||K$);
3) догадаться, что **Hash**($M_2||K$) точно такой же, и тем самым подделать действительный жетон и нарушить безопасность MAC.

Построение HMAC

Построение имитовставки на основе функции хеширования (hash-based MAC – HMAC) позволяет получить MAC по хеш-функции, что более безопасно, чем секретный префикс или суффикс. HMAC является безопасной PRF при условии, что хеш-функция стойка к коллизиям, но даже если это не так, HMAC все же дает безопасную PRF,

если функция сжатия внутри хеш-функции псевдослучайная. Во всех безопасных протоколах связи – IPSec, SSH и TLS – используется HMAC. (Спецификации HMAC приведены в стандарте NIST FIPS 198-1 и в документе RFC 2104.)

В HMAC для вычисления аутентификационного жетона используется функция хеширования, **Hash**, показанная на рис. 7.1 и определяемая следующим выражением:

$$\textbf{Hash}((K \oplus opad) \| \textbf{Hash}((K \oplus ipad) \| M)).$$

Здесь *opad* (внешнее дополнение) – строка вида (*5c5c5c…5c*), длина которой равна длине блока **Hash**. Ключ *K* обычно короче одного блока, он дополняется байтами 00 и объединяется с *opad* операцией XOR. Например, если *K* – однобайтовая строка *00*, то $K \oplus opad = opad$ (то же самое верно, если *K* – строка произвольной длины, не превышающей длину блока, состоящая из одних нулей). $K \oplus opad$ – первый блок, обрабатываемый внешним вызовом **Hash**, а именно левым вызовом **Hash** в приведенном выше выражении, или нижним хешированием на рис. 7.1.

ipad (внутреннее дополнение) – строка вида (*363636…36*), длина которой равна длине блока **Hash** и которая тоже дополняется байтами 00. Результирующий блок – первый блок, обрабатываемый внутренним вызовом **Hash**, а именно правым вызовом **Hash** в выражении выше, или верхним хешированием на рис. 7.1.

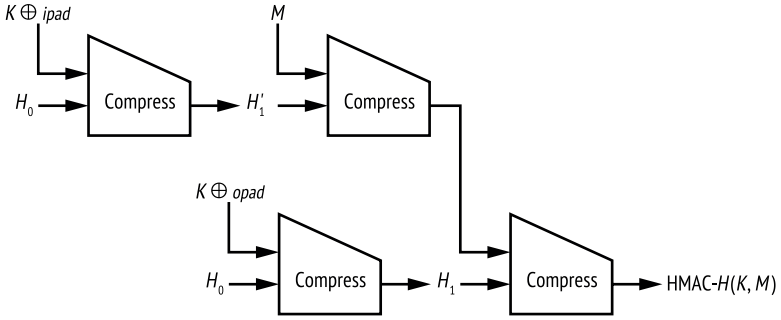

Рис. 7.1. Построение имитовставки на основе функции хеширования

Примечание *Метод конверта – более безопасное построение, чем секретный префикс или суффикс. Он описывается выражением* **Hash**$(K \| M \| K)$*, иногда называемым бутербродной имитовставкой, но теоретически менее безопасен, чем HMAC.*

Если в роли **Hash** выступает хеш-функция SHA-256, то такой экземпляр HMAC называется HMAC-SHA-256. Вообще, HMAC-*Hash* называется экземпляр HMAC с хеш-функцией *Hash*. Это значит, что если вас попросят использовать HMAC, то следует спросить: «С какой хеш-функцией?»

Обобщенная атака против MAC на основе функций хеширования

Существует одна атака, которая работает против всех имитовставок, основанных на повторении функции хеширования. Вспомните атаку, описанную в разделе «Построение секретного суффикса» выше, где коллизия хешей использовалась для получения коллизии имитовставок. Ту же стратегию можно применить для атаки на MAC с секретным префиксом или на HMAC, хотя последствия и не такие печальные.

Для иллюстрации этой атаки рассмотрим MAC с секретным префиксом **Hash**($K||M$), показанную на рис. 7.2. Если длина дайджеста равна n бит, то можно найти два сообщения M_1 и M_2 такие, что **Hash**($K||M_1$) = **Hash**($K||M_2$), запросив примерно $2^{n/2}$ жетонов у системы, хранящей ключ (вспомните атаку на основе парадокса дней рождения, описанную в главе 6). Если хеш-функция уязвима для атаки удлинением сообщения, как SHA-256, то M_1 и M_2 можно использовать для подделки MAC, для чего следует выбрать какие-то произвольные данные M_3 и запросить у оракула MAC значение **Hash**($K||M_1||M_3$), которое будет являться имитовставкой сообщения $M_1||M_3$. Как выясняется, это также имитовставка сообщения $M_2||M_3$, поскольку внутреннее состояние хеша M_1 и M_3 такое же, как хеша M_2 и M_3. Таким образом, мы успешно подделали MAC-жетон. (Атака становится неосуществимой, когда n больше, скажем, 128 бит.)

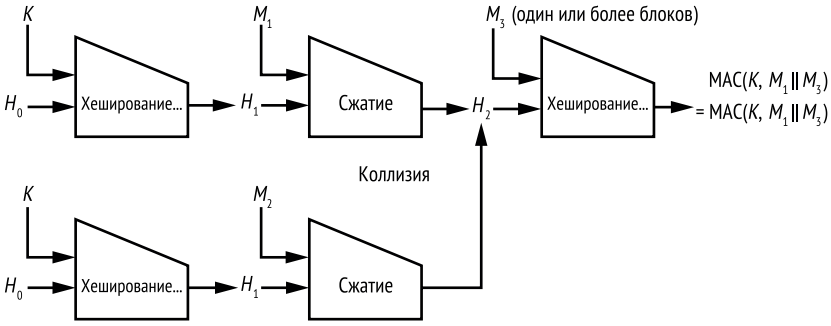

Рис. 7.2. Принцип обобщенной атаки с подделкой на MAC на основе функций хеширования

Эта атака работает, даже если хеш-функция не уязвима к атаке удлинением сообщения, и против HMAC она тоже применима. Стоимость атаки зависит от размера цепного значения и длины MAC: если цепное значение MAC имеет длину 512 бит, а жетоны – 128 бит, то после 2^{64} вычислений будет найдена коллизия MAC, но вряд ли коллизия во внутреннем состоянии, потому что для нахождения такой коллизии потребовалось бы в среднем $2^{512/2} = 2^{256}$ операций.

Создание функций хеширования на основе блочных шифров: CMAC

Напомним (см. главу 6), что функции сжатия во многих хеш-функциях построены на основе блочных шифров. Например, псевдослучайная функция HMAC-SHA-256 представляет собой последовательность обращений к функции сжатия SHA-256, которая сама является блочным шифром с повторяющейся последовательностью раундов. Иными словами, HMAC-SHA-256 – это блочный шифр внутри хеш-функции, применяемой в построении HMAC. Так почему бы не использовать сам блочный шифр вместо такой многоуровневой конструкции?

CMAC (cipher-based MAC – имитовставка на основе шифра) как раз и является такого рода построением: оно создает MAC, пользуясь только блочным шифром, например AES. Хотя CMAC не столь популярна, как HMAC, она встречается во многих системах, включая протокол обмена ключей в интернете (Internet Key Exchange – IKE), являющийся частью набора протоколов IPSec. Например, IKE генерирует материал для ключей, применяя построение AES-CMACPRF-128 в качестве базового алгоритма (или CMAC на основе AES со 128-битовым выходом). Построение CMAC специфицировано в RFC 4493.

Взлом CBC-MAC

CMAC была спроектирована в 2005 году как усовершенствованная версия *CBC-MAC*, более простой имитовставки на основе блочного шифра, работающего в режиме сцепления блоков (CBC) (см. раздел «Режимы работы» главы 4).

CBC-MAC, предшественница CMAC, устроена просто: чтобы вычислить жетон сообщения M при заданном блочном шифре $\mathbf{E}$, следует зашифровать M в режиме CBC с нулевым начальным значением (IV) и отбросить все, кроме последнего блока шифртекста. То есть мы вычисляем $C_1 = \mathbf{E}(K, M_1)$, $C_2 = \mathbf{E}(K, M_2 \oplus C_1)$, $C_3 = \mathbf{E}(K, M_3 \oplus C_2)$ и т. д. для каждого блока M и оставляем только последнее значение C_i, которое и является CBC-MAC-жетоном для M. Легко вычислить – и легко атаковать.

Чтобы понять, почему CBC-MAC небезопасен, рассмотрим CBC-MAC-жетон $T_1 = \mathbf{E}(K, M_1)$ одноблочного сообщения M_1 и жетон $T_2 = \mathbf{E}(K, M_2)$ другого одноблочного сообщения M_2. Зная обе пары (M_1, T_1) и (M_2, T_2), мы можем заключить, что T_2 является также жетоном двухблочного сообщения $M_1 \| (M_2 \oplus T_1)$. Действительно, если применить CBC-MAC к $M_1 \| (M_2 \oplus T_1)$ и вычислить $C_1 = \mathbf{E}(K, M_1) = T_1$, а затем $C_2 = \mathbf{E}(K, (M_2 \oplus T_1) \oplus T_1) = \mathbf{E}(K, M_2) = T_2$, то можно создать третью пару сообщение–жетон из двух таких пар, не зная ключа. То есть мы можем подделать CBC-MAC-жетоны, а следовательно, нарушить безопасность CBC-MAC.

Исправление CBC-MAC

CMAC исправляет CBC-MAC, поскольку обрабатывает последний блок, используя не тот ключ, что при обработке предшествующих блоков.

Для этого CMAC сначала формирует два ключа K_1 и K_2 на основе главного ключа K, так что K, K_1 и K_2 различаются. В CMAC последний блок обрабатывается с применением K_1 или K_2, а предыдущие – с применением K.

Чтобы сформировать K_1 и K_2, CMAC сначала вычисляет временное значение $L = \mathbf{E}(0, K)$, где 0 выступает в роли ключа блочного шифра, а K в роли блока открытого текста. Затем CMAC полагает K_1 равным $(L << 1)$, если старший бит L равен 0, или $(L << 1) \oplus 87$, если старший бит L равен 1 (число 87 выбрано за его математические свойства в случае, когда длина блока данных равна 128 бит; если длина блока иная, то нужно выбирать другое число).

Значение K_2 полагается равным $(K_1 << 1)$, если старший бит K_1 равен 0, или $K_2 = (K_1 << 1) \oplus 87$ в противном случае.

При известных K_1 и K_2 CMAC работает, как CBC-MAC, для всех блоков, кроме последнего. Если длина последнего куска сообщения M_n в точности равна длине блока, то CMAC возвращает в качестве жетона значение $\mathbf{E}(K, M_n \oplus C_{n-1} \oplus K_1)$, как показано на рис. 7.3. Но если M_n содержит меньше битов, чем полный блок, то CMAC дополняет его одним единичным битом и нулями и возвращает в качестве жетона $\mathbf{E}(K, M_n \oplus C_{n-1} \oplus K_2)$, как показано на рис. 7.4. Заметим, что в первом случае используется только K_1, а во втором только K_2, но в обоих случаях для обработки всех блоков, кроме последнего, применяется главный ключ K.

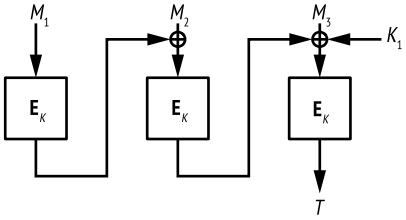

Рис. 7.3. Построение имитовставки CMAC на основе блочного шифра в случае, когда сообщение точно делится на блоки равной длины

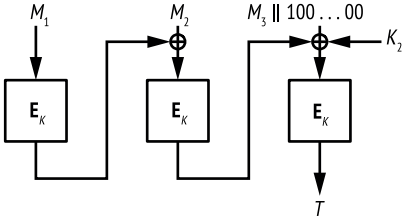

Рис. 7.4. Построение имитовставки CMAC на основе блочного шифра в случае, когда последний кусок сообщения дополняется одним единичным битом и нулями до полного блока

Заметим, что, в отличие от режима работы CBC, CMAC не принимает параметра IV и детерминирована: CMAC всегда возвращает один и тот же жетон для данного сообщения M, потому что вычисление **CMAC**(M) не рандомизировано. И это хорошо, т. к., в отличие от шифрования, вычисление MAC не обязано быть рандомизированным, чтобы считаться безопасным, что снимает с нас бремя выбора случайного IV.

Проектирование специализированных имитовставок

Мы видели, как можно использовать функции хеширования и блочные шифры для построения PRF, являющихся безопасными при условии, что безопасны базовая хеш-функция или шифр. В схемах типа HMAC или CMAC просто комбинируются имеющиеся хеш-функции или блочные шифры для получения безопасных PRF или MAC. Повторное использование алгоритмов – это удобно, но всегда ли это самый эффективный подход?

Интуитивно кажется, что PRF и MAC должны требовать меньше работы, чем хеш-функции без ключа, чтобы считаться безопасными, – ведь включение в них секретного ключа мешает не знающему его противнику экспериментировать с алгоритмом. Кроме того, PRF и MAC раскрывают противнику только короткий жетон, в отличие от блочных шифров, которые порождают шифртекст такой же длины, как исходное сообщение. Поэтому PRF и MAC, по идее, не нужна вся мощь хеш-функций или блочных шифров. Это соображение легло в основу специализированных алгоритмов, предназначенных только для вычисления PRF и/или MAC.

В последующих разделах мы рассмотрим два таких алгоритма, получивших широкое распространение: Poly1305 и SipHash. Я объясню принципы их проектирования и почему они, скорее всего, безопасны.

Poly1305

Алгоритм Poly1305 (произносится «поли тринадцать-ноль-пять») был спроектирован в 2005 году Дэниелем Дж. Бернштейном (автором потокового шифра Salsa20, который обсуждался в главе 5, и шифра ChaCha, легшего в основу хеш-функций BLAKE и BLAKE2, рассмотренных в главе 6). Poly1305 оптимизирован для работы на современных процессорах и на момент написания этой книги используется, в частности, Google для защиты HTTPS-соединений (HTTP поверх TLS) и в пакете OpenSSH. В отличие от Salsa20, дизайн Poly1305 основан на методах, восходящих к 1970-м годам, а именно на универсальных функциях хеширования и построении Вегмана–Картера.

Универсальные функции хеширования

В имитовставке Poly1305 используется *универсальная функция хеширования*, которая значительно слабее криптографических хеш-функций, но вместе с тем и значительно быстрее. Например, универсальные функции хеширования не обязаны быть стойкими к коллизиям. А это значит, что для для достижения целей им нужно выполнить меньше работы.

Как и PRF, универсальная функция хеширования параметризуется секретным ключом: если даны сообщение M и ключ K, то $\mathbf{UH}(K, M)$ обозначает выход универсальной функции хеширования **UH**. К уни-

версальной функции хеширования предъявляется лишь одно требование безопасности: для любых двух сообщений M_1 и M_2 вероятность того, что $\mathbf{UH}(K, M_1) = \mathbf{UH}(K, M_2)$, должна быть пренебрежимо мала для случайного ключа K. В отличие от PRF, универсальная функция хеширования не обязана быть псевдослучайной; просто не должно существовать пары (M_1, M_2), которая дает один и тот же хеш для многих разных ключей. Поскольку удовлетворить таким требованиям проще, требуется меньше операций, и потому универсальные функции хеширования работают значительно быстрее PRF.

Однако использовать универсальный хеш в качестве MAC можно для аутентификации максимум одного сообщения. Рассмотрим, к примеру, универсальную функцию хеширования, используемую в алгоритме Poly1305: функцию *полиномиального вычисления*. (См. основополагающую статью Gilbert, MacWilliams, and Sloane «Codes Which Detect Deception», вышедшую в 1974 году, где приведены дополнительные сведения на эту тему.) Такого рода хеш-функции параметризуются простым числом p и принимают ключ, состоящий из двух чисел R и K в диапазоне $[1, p]$, и сообщение M, состоящее из n блоков, $(M_1, M_2, \ldots, M_n)$. Тогда результат универсальной функции хеширования вычисляется следующим образом:

$$\boldsymbol{UH}(R, K, M) = R + M_1K + M_2K^2 + M_3K^3 + \ldots + M_nK^n \bmod p.$$

Знаком + обозначено сложение положительных целых чисел, K^i – результат возведения K в степень i, а «mod p» – операция деления по модулю p (т. е. остаток от деления на p, например $12 \bmod 10 = 2$, $10 \bmod 10 = 0$, $8 \bmod 10 = 8$ и т. д.).

Поскольку мы хотим вычислять хеш максимально быстро, имитовставки на основе универсальной хеш-функции часто работают с блоками длиной 128 бит и простым числом p, ненамного большим 2^{128}, например $2^{128} + 51$. При длине 128 можно добиться очень быстрой реализации благодаря эффективному использованию 32- и 64-разрядных АЛУ типичных процессоров.

Потенциальные уязвимости

У универсальных функций хеширования есть одна слабость: поскольку универсальный хеш может безопасно аутентифицировать только одно сообщение, противник мог бы взломать показанную выше MAC на основе полиномиального вычисления, запросив жетоны всего двух сообщений. Точнее, он мог бы запросить жетоны для сообщения с $M_1 = M_2 = \ldots = 0$ (для него жетон $\mathbf{UH}(R, K, 0) = R$), а затем использовать их для нахождения секретного значения R. Или противник мог бы запросить жетоны для сообщения с $M_1 = 1$, $M_2 = M_3 = \ldots = 0$ (его жетон равен $T = R + K$), это позволило бы ему найти K, вычтя R из T. Теперь противник знает весь ключ (R, K) и может подделывать имитовставки для любого сообщения.

По счастью, существует способ перейти от безопасности для одного сообщения к безопасности для нескольких сообщений.

Имитовставки Вегмана–Картера

Способ аутентификации нескольких сообщений с помощью универсальной функции хеширования придумали исследователи из IBM Вегман и Картер и описали в статье «New Hash Functions and Their Use in Authentication and Set Equality», опубликованной в 1981 году. В построении Вегмана–Картера MAC строится с помощью универсальной хеш-функции и PRF с использованием двух ключей, K_1 и K_2, по формуле

$$\mathbf{MAC}(K_1, K_2, N, M) = \mathbf{UH}(K_1, M) + \mathbf{PRF}(K_2, N),$$

где N – одноразовое число, уникальное для каждого ключа K_2, а длина выхода **PRF** совпадает с длиной результата универсальной функции хеширования **UH**. За счет сложения обоих значений стойкая псевдослучайная функция **PRF** маскирует криптографическую слабость **UH**. Это можно рассматривать как шифрование результата универсальной функции хеширования, где PRF играет роль потокового шифра и предотвращает описанную выше атаку, делая возможной аутентификацию нескольких сообщений одним и тем же ключом K_1.

Подведем итог: построение Вегмана–Картера $\mathbf{UH}(K_1, M) + \mathbf{PRF}(K_2, N)$ дает безопасную MAC при следующих предположениях:

- **UH** – безопасная универсальная функция хеширования;
- **PRF** – безопасная PRF;
- каждое одноразовое число N используется ровно один раз для каждого ключа K_2;
- выходные значения **UH** и **PRF** достаточно длинные, чтобы обеспечить надлежащую безопасность.

Теперь посмотрим, как в алгоритме Poly1305 построение Вегмана–Картера используется для создания безопасной и быстрой MAC.

Poly1305-AES

Алгоритм Poly1305 первоначально был предложен как Poly1305-AES, т. е. универсальная функция хеширования Poly1305 сочеталась с блочным шифром AES. Poly1305-AES гораздо быстрее MAC на основе HMAC и даже CMAC, потому что вычисляет только один блок AES и обрабатывает сообщение параллельно, выполняя серию простых арифметических операций.

Если даны 128-битовые K_1, K_2, N и сообщение M, то Poly1305-AES возвращает следующее выражение:

$$\mathbf{Poly1305}(K_1, M) + \mathrm{AES}(K_2, N) \bmod 2^{128}.$$

Деление на 2^{128} гарантирует, что результат поместится в 128 бит. Сообщение M обрабатывается как последовательность 128-битовых блоков (M_1, M_2, …, M_n), и в конец каждого блока дописывается 129-й бит,

так чтобы сделать все блоки 129-битовыми. (Если последний блок короче 16 байт, то он дополняется одним единичным битом и последующими нулевыми битами перед добавлением последнего 129-го бита.) Затем Poly1305 вычисляет следующий полином:

$$\textbf{Poly1305}(K_1, M) = M_1K_1^n + M_2K_1^n + \ldots + M_nK_1 \bmod 2^{130} - 5.$$

Результатом вычисления является целое число длиной не более 129 бит. После прибавления 128-битового значения $\textbf{AES}(K_2, N)$ результат по модулю 2^{128} и принимается в качестве 128-битовой имитовставки.

Примечание *AES представляет собой не PRF, а псевдослучайную перестановку (PRP). Однако здесь это не важно, потому что построение Вегмана–Картера работает с PRP точно так же, как с PRF, поскольку если задана функция, которая является либо PRF, либо PRP, трудно определить, что именно она собой представляет, глядя только на ее выходные значения.*

Анализ безопасности Poly1305-AES (см. статью «The Poly1305-AES Message-Authentication Code» по адресу *http://cr.yp.to/mac/poly1305-20050329.pdf*) показывает, что уровень безопасности Poly1305-AES равен 128 бит, при условии что AES – безопасный блочный шифр и, конечно, что все реализовано корректно, но последнее относится к любому криптографическому алгоритму.

Универсальную функцию хеширования Poly1305 можно комбинировать с другими алгоритмами, а не только с AES. Например, Poly1305 использовалась с потоковым шифром ChaCha (см. RFC 7539, «ChaCha20 and Poly1305 for IETF Protocols»). Без сомнения, Poly1305 будет и дальше использоваться, когда необходима быстрая MAC.

SipHash

Алгоритм Poly1305 быстрый и безопасный, но у него есть несколько недостатков. Во-первых, полиномиальное вычисление трудно реализовать эффективно, особенно если не владеешь соответствующим математическим аппаратом (примеры см. по адресу *https://github.com/floodyberry/poly1305-donna/*). Во-вторых, он безопасен только для одного сообщения, если не использовать построение Вегмана–Картера. А если использовать, то необходимо одноразовое число, и алгоритм перестает быть безопасным, если это число применяется несколько раз. Наконец, Poly1305 оптимизирован для длинных сообщений, а при обработке коротких (скажем, меньше 128 байт) становится стрельбой из пушек по воробьям. В таких случаях решением является алгоритм SipHash.

Я спроектировал SipHash в 2012 году совместно с Дэном Бернштейном, первоначально ставилась цель решить некриптографическую проблему: атаку на хеш-таблицы, вызывающую отказ от обслужи-

вания. Хеш-таблицы – это структуры данных, применяемые в языках программирования для эффективного хранения элементов. До появления SipHash хеш-таблицы полагались на некриптографические хеш-функции с ключом, для которых было легко найти коллизии, а значит, эксплуатировать удаленную систему, замедлив доступ к хеш-таблице и тем самым реализовав DoS-атаку. Мы установили, что использование PRF решило бы эту проблему, и начали проектировать псевдослучайную функцию SipHash, подходящую для хеш-таблиц. Поскольку хеш-таблицы обрабатывают в основном короткие данные, SipHash оптимизирована для коротких сообщений. Однако SipHash можно использовать не только для хеш-таблиц, это полноценная PRF и MAC, достоинства которой в полной мере раскрываются в ситуациях, когда входные данные преимущественно короткие.

Как работает SipHash

В SipHash применяется трюк, благодаря которому она оказывается более безопасной, чем простые функции губки: вместо того чтобы применять к блокам XOR только один раз перед перестановкой, SipHash применяет XOR до и после перестановки, как показано на рис. 7.5. 128-битовый ключ SipHash рассматривается как два 64-битовых слова, K_1 и K_2, которые с помощью XOR объединяются в 256-битовое фиксированное начальное состояние, рассматриваемое как четыре 64-битовых слова. Затем ключи отбрасываются, и вычисление SipHash сводится к итеративному выполнению базовой функции SipRound с последующим применением XOR к блокам сообщения с целью модификации четырехсловного внутреннего состояния. Наконец, SipHash возвращает 64-битовый жетон, объединяя четыре слова состояния с помощью XOR.

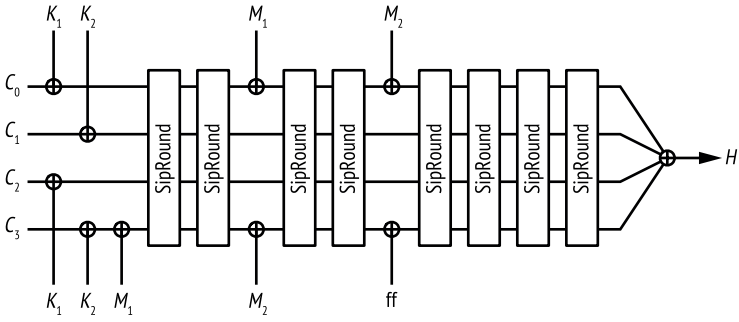

Рис. 7.5. Обработка 15-байтового сообщения в алгоритме SipHash-2-4 (блок M_1 длиной 8 байт и блок M_2 длиной 7 байт плюс 1 байт дополнения)

Функция SipRound сочетает операции XOR со сложениями и циклическими сдвигами слова, чтобы добиться безопасности. Она преобразует состояние – четыре 64-битовых слова (a, b, c, d), – выполняя следующие операции в порядке сверху вниз. Операции в левом и правом столбцах независимы, поэтому могут выполняться параллельно.

a += b $\quad$ c += d
b <<<= 13 $\quad$ d <<<= 16
b ⊕= a $\quad$ d ⊕= c
a <<<= 32
c += b $\quad$ a += d
b <<<= 17 $\quad$ d <<<= 21
b ⊕= c $\quad$ d ⊕= a
c <<<= 32

Здесь a += b – сокращенная запись $a = a + b$, а b <<< = 13 – сокращенная запись $b = b$ <<< 13 (64-битовое слово b циклически сдвигается влево на 13 бит).

Эти простые операции над 64-битовыми словами – почти все, что необходимо для вычисления SipHash, хотя вовсе необязательно делать это самостоятельно. Готовые реализации имеются на большинстве языков, включая C, Go, Java, JavaScript и Python.

Примечание *Говоря о версии SipHash, пишут **SipHash**-x-y, где x – количество раундов перед подачей очередного блока сообщения, а y – количество раундов после подачи. Чем больше раундов, тем больше операций, поэтому работа функции замедляется, зато повышается безопасность. По умолчанию подразумевается SipHash-2-4 (обозначается просто SipHash), и даже она до сих пор она не поддавалась криптоанализу. Но можете перестраховаться и выбрать версию SipHash-4-8, которая выполняет в два раза больше операций и, стало быть, в два раза медленнее.*

Какие возможны проблемы

Подобно шифрам и хеш-функциям без ключа, MAC и PRF, безопасные на бумаге, могут оказаться уязвимы к атакам при использовании в полевой обстановке, когда атакует реальный противник. Рассмотрим два примера.

Атаки с хронометражем на верификацию MAC

Атаки по побочным каналам направлены против реализации криптографического алгоритма, а не против самого алгоритма. В частности, в *атаках с хронометражем* замеряется время выполнения алгоритма, чтобы получить секретную информацию: о ключах, открытом тексте и секретных случайных значениях. Легко представить себе, что сравнение строк, занимающее переменное время, несет в себе уязвимость, и это справедливо не только для верификации MAC, но и для многих других криптографических алгоритмов и средств обеспечения безопасности.

MAC может быть уязвима к атакам с хронометражем, если время проверки жетонов удаленной системой зависит от значения жетона, что позволяет противнику определить правильный жетон сообще-

ния, многократно отправляя неправильные, – правильным будет тот, для проверки которого нужно наибольшее время. Проблема возникает, если сервер сравнивает строки правильного и неправильного жетонов побайтно, прекращая сравнение, как только встретятся два разных байта. Например, в листинге 7.1 показан код сравнения двух строк на Python, занимающий переменное время: если различаются первые байты, то функция возвращает управление после всего одного сравнения, а если строки x и y совпадают, то количество сравнений будет равно длине строки.

Листинг 7.1. Сравнение двух n-байтовых строк, занимающее переменное время

```
def verify_mac(x, y, n):
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True
```

Чтобы продемонстрировать уязвимость функции `verify_mac()`, напишем программу, которая измеряет время выполнения 100 000 вызовов, сначала с одинаковыми 10-байтовыми значениями x и y, а затем со значениями, различающимися в третьем байте. Мы ожидаем, что второе сравнение потребует заметно больше времени, чем первое, потому `verify_mac()` будет сравнивать меньше байтов.

Листинг 7.2. Измерение времени при выполнении функции `verify_mac()` из листинга 7.1

```
from time import time

MAC1 = '0123456789abcdef'
MAC2 = '01X3456789abcdef'
TRIALS = 100000

# при каждом обращении к verify_mac() просматриваются все восемь байт
start = time()
for i in range(TRIALS):
    verify_mac(MAC1, MAC1, len(MAC1))
end = time()
print("%0.5f" % (end-start))

# при каждом обращении к verify_mac() просматриваются только три байта
start = time()
for i in range(TRIALS):
  verify_mac(MAC1, MAC2, len(MAC1))
end = time()
print("%0.5f" % (end-start))
```

На моей машине в результате типичного выполнения этой программы печаталось время приблизительно 0.215 и 0.095 секунды со-

ответственно. Разница достаточно велика, чтобы понять, что происходит внутри алгоритма. Аналогичные различия во времени можно наблюдать, когда первый несовпадающий байт находится в другой позиции. Если `MAC1` – правильный MAC-жетон, а `MAC2` – жетон, опробованный противником, то легко определить позицию первого несовпадения, равную числу правильно угаданных байтов.

Разумеется, если время выполнения не зависит от секретного хронометража, то такая атака работать не будет, поэтому опытные разработчики стремятся к реализациям *с постоянным временем* – когда код выполняется одинаковое время при любом входном значении секрета. Например, функция на C в листинге 7.3 сравнивает два буфера размером `size` байт за постоянное время: временная переменная `result` будет отлична от нуля тогда и только тогда, когда буферы различаются хотя бы в одной позиции.

Листинг 7.3. Сравнение двух буферов за постоянное время дает более безопасную верификацию MAC

```
int cmp_const(const void *a, const void *b, const size_t size)
{
    const unsigned char *_a = (const unsigned char *) a;
    const unsigned char *_b = (const unsigned char *) b;
    unsigned char result = 0;
    size_t i;

    for (i = 0; i < size; i++) {
        result |= _a[i] ^ _b[i];
    }

    return result; /* возвращает 0, если *a и *b равны, иначе ненулевое значение */
}
```

Когда губки протекают

Алгоритмы на основе перестановок, в частности SHA-3 и SipHash, просты, легко реализуются и имеют компактный код, но они уязвимы к атакам по побочным каналам, которые восстанавливают снимок состояния системы. Например, если процесс может в любой момент времени прочитать оперативную память и значения регистров или дамп памяти, то противник способен определить внутреннее состояние SHA-3 в режиме MAC или внутреннее состояние SipHash, а затем вычислить обратную перестановку и восстановить начальное секретное состояние. После этого он может подделать жетон любого сообщения, нарушив тем самым безопасность MAC.

По счастью, эта атака бессильна против имитовставок на основе функций сжатия, в частности HMAC-SHA-256 и BLAKE2 с секретным ключом, потому что противнику понадобился бы снимок памяти точно в тот момент, когда используется ключ. Вывод: если вы работаете в среде, где части памяти процесса могут утекать наружу, то следу-

ет использовать MAC, основанную на необратимой функции сжатия, а не на перестановке.

Для дополнительного чтения

Достопочтенная HMAC заслуживает больше внимания, чем я могу уделить в этой книге, особенно чтобы осветить ход мыслей, который сначала привел к ее широкому признанию, а затем к отказу в случае комбинации со слабой хеш-функцией. Я рекомендую статью Bellare, Canetti, and Krawczyk «Keying Hash Functions for Message Authentication» 1996 года, в которой впервые описана HMAC и ее близкая родственница NMAC, а также ее продолжение – статью Bellare «New Proofs for NMAC and HMAC: Security Without Collision-Resistance» 2006 года, в которой доказывается, что HMAC не нуждается в хеш-функции, стойкой к коллизиям, а лишь в хеш-функции, для которой функция сжатия псевдослучайная. С другой стороны, в статье Fouque, Leurent, and Nguyen «Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5» 2007 года показано, как атаковать HMAC и NMAC, основанные на уязвимой хеш-функции типа MD4 или MD5. (Кстати говоря, нельзя сказать, что HMAC-MD5 и HMAC-SHA-1 совсем никуда не годятся, но риск достаточно велик.)

Имитовставки Вегмана–Картера также достойны более пристального внимания как в силу практического интереса, так и благодаря лежащей в их основе теории. Основополагающие работы Вегмана и Картера можно скачать по адресу *http://cr.yp.to/bib/entries.html*. Из числа более современных проектов упомянем UMAC и VMAC – одни из самых быстрых MAC для длинных сообщений.

В этой главе не обсуждалась MAC *Pelican*, в которой блочный шифр AES, сокращенный до четырех раундов (вместо 10 в полной версии шифра), используется для аутентификации блоков сообщений с помощью довольно простого построения, описанного по адресу *https://eprint.iacr.org/2005/088/*. Впрочем, Pelican – скорее, курьез, который на практике применяется редко.

И последнее: если вас интересуют поиск уязвимостей в криптографическом ПО, поищите места, где используется CBC-MAC, или слабости, вызванные обработкой в HMAC ключей произвольного размера – взятие в качестве ключа **Hash**(*K*) вместо *K*, если *K* слишком длинный, в результате чего *K* и **Hash**(*K*) становятся *эквивалентными ключами*. Или просто поищите системы, которые не используют MAC, хотя должны бы, – довольно частая ситуация.

В главе 8 мы рассмотрим, как можно объединить MAC с шифрами для защиты подлинности, целостности *и* конфиденциальности сообщения. Мы также обсудим, как это можно сделать без MAC благодаря шифрам с аутентификацией, которые сочетают функциональность базового шифра с функциональностью MAC, возвращая жетон вместе с каждым шифртекстом.

8

ШИФРОВАНИЕ С АУТЕНТИФИКАЦИЕЙ

Эта глава посвящена типу алгоритмов, защищающих не только конфиденциальность, но и подлинность сообщения. Напомним (см. главу 7), что имитовставки (MAC) – это алгоритмы, защищающие подлинность сообщения посредством создания жетона, своего рода подписи. Как и MAC, алгоритмы шифрования с аутентификацией (authenticated encryption – AE), обсуждаемые в этой главе, порождают аутентификационный жетон, но при этом еще и шифруют сообщение. Иными словами, один алгоритм AE предлагает функциональность, характерную для комбинации обычного шифра и MAC.

Сочетание шифра и MAC может обеспечить различные уровни шифрования с аутентификацией, как мы узнаем на протяжении этой главы. Я рассмотрю несколько способов комбинирования MAC с шифрами, объясню, какие методы самые безопасные, и расскажу о шиф-

рах, порождающих как шифртекст, так и аутентификационный жетон. Затем мы рассмотрим четыре важных шифра с аутентификацией: три построения на основе блочных шифров с упором на популярный шифр AES в режиме счетчика с аутентификацией Галуа (AES-GCM) и один шифр, в котором используется только алгоритм перестановки.

Шифрование с аутентификацией с использованием MAC

Как показано на рис. 8.1, MAC и шифры можно сочетать одним из трех способов: шифрование-и-MAC, MAC-затем-шифрование и шифрование-затем-MAC.

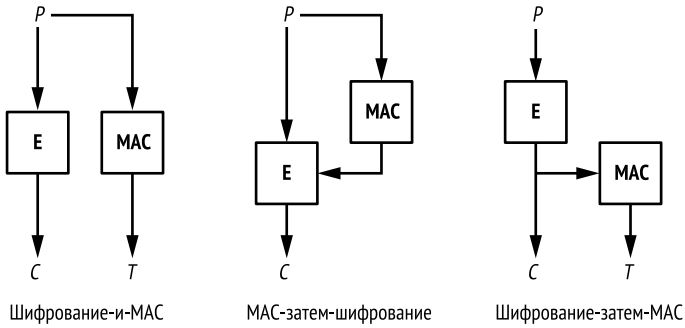

Рис. 8.1. Комбинации шифра и MAC

Эти три комбинации различаются порядком применения шифрования и генерирования аутентификационного жетона. Но выбор конкретного алгоритма MAC или шифра не играет роли, при условии что каждый безопасен и ключи MAC и шифра различны.

По рис. 8.1 видно, что в схеме шифрование-и-MAC открытый текст зашифровывается и аутентификационный жетон генерируется по открытому тексту, так что обе операции (шифрование и аутентификация) не зависят друг от друга и могут вычисляться параллельно. В схеме MAC-затем-шифрование сначала генерируется жетон по открытому тексту, а затем открытый текст и MAC зашифровываются совместно. Наоборот, в схеме шифрование-затем-MAC сначала зашифровывается открытый текст, а затем по шифртексту генерируется жетон.

Потребление ресурсов во всех трех подходах приблизительно одинаково. Поговорим о том, какой из них наиболее безопасен.

Шифрование-и-MAC

В схеме *шифрование-и-MAC* шифртекст и MAC-жетон вычисляются по отдельности. Зная открытый текст (P), отправитель вычисляет шифртекст $C = \mathbf{E}(K_1, P)$, где $\mathbf{E}$ – алгоритм шифрования, а C – результирующий

шифртекст. Аутентификационный жетон (T) вычисляется по открытому тексту в виде $T = \mathbf{MAC}(K_2, P)$. C и T можно вычислять как последовательно, так и параллельно.

Сгенерировав шифртекст и аутентификационный жетон, отправитель передает то и другое получателю. Получатель дешифрирует C и получает открытый текст $P = \mathbf{D}(K_1, C)$. Затем он вычисляет $\mathbf{MAC}(K_2, P)$ по дешифрированному открытому тексту и сравнивает результат с полученным T. Проверка завершается неудачно, если C или T повреждены, в этом случае сообщение признается недействительным.

По крайней мере в теории, шифрование-и-MAC – наименее безопасное сочетание MAC и шифра, потому что даже из безопасной MAC может утекать информация о P, и тогда P будет легче восстановить. Поскольку цель MAC – всего лишь не допустить подделки жетонов, а жетоны необязательно выглядят случайными, аутентификационный жетон (T) открытого текста (P) все же может давать утечку информации, пусть даже MAC считается безопасной! (Разумеется, если MAC – псевдослучайная функция, то жетон ничего не выдаст о P.)

Но, несмотря на относительную слабость, схема шифрование-и-MAC по-прежнему поддерживается многими системами, в т. ч. безопасным протоколом транспортного уровня SSH, где каждый зашифрованный пакет C сопровождается жетоном $T = \mathbf{MAC}(K, N||P)$, который передается в виде незашифрованного пакета P. В этом выражении N – 32-битовый порядковый номер, который увеличивается для каждого отправленного пакета, чтобы гарантировать обработку принятых пакетов в правильном порядке. На практике схема шифрование-и-MAC оказалась достаточно хорошей в SSH благодаря использованию стойких алгоритмов формирования MAC типа HMAC-SHA-256, которые не дают утечки информации о P.

MAC-затем-шифрование

Сочетание *MAC-затем-шифрование* защищает сообщение P, поскольку сначала вычисляется аутентификационный жетон $T = \mathbf{MAC}(K_2, P)$. Затем создается шифртекст путем совместного зашифровывания открытого текста и жетона: $C = \mathbf{E}(K_1, P||T)$.

Завершив эти действия, отправитель передает только сообщение C, содержащее зашифрованные открытый текст и жетон. Получатель дешифрирует C, вычисляя $P||T = \mathbf{D}(K_1, C)$, и получает открытый текст и жетон T. Затем получатель проверяет полученный жетон T, вычисляя по открытому тексту $\mathbf{MAC}(K_2, P)$, сравнивает вычисленный жетон с полученным.

Как и в случае схемы шифрование-и-MAC, при использовании схемы MAC-затем-шифрование получатель должен дешифрировать C, прежде чем сумеет определить, были ли повреждены пакеты, – таким образом он выявляет потенциально поврежденные открытые тексты. Тем не менее схема MAC-затем-шифрование безопаснее, чем шифрование-и-MAC, потому что скрывает аутентификационный жетон

открытого текста и тем самым предотвращает утечку информации об открытом тексте.

Схема MAC-затем-шифрование много лет использовалась в протоколе TLS, но в версии TLS 1.3 была заменена шифрами с аутентификацией (подробнее о TLS 1.3 см. главу 13).

Шифрование-затем-MAC

В схеме шифрование-затем-MAC получателю отправляется два значения: шифртекст $C = \mathbf{E}(K_1, P)$ и вычисленный по шифртексту жетон $T = \mathbf{MAC}(K_2, C)$. Получатель вычисляет жетон $\mathbf{MAC}(K_2, C)$ и проверяет, совпадает ли он с полученным жетоном T. Если значения одинаковы, то вычисляется открытый текст $P = \mathbf{D}(K_1, C)$, в противном случае шифртекст отбрасывается.

Преимущество этого метода заключается в том, что для обнаружения поврежденных сообщений получателю нужно вычислить только MAC, а дешифрировать поврежденный шифртекст нет необходимости. Еще одно преимущество – то, что противник не может отправить получателю пару C и T для дешифрирования, не взломав MAC, поэтому противнику труднее передать вредоносные данные.

Благодаря такому сочетанию особенностей схема шифрование-затем-MAC более стойкая, чем две другие. Именно по этой причине в широко используемом наборе протоколов IPSec она и применяется для защиты пакетов (например, в VPN-туннелях).

Но почему тогда эта схема не используется в протоколах SSH и TLS? Ответ прост: когда создавались SSH и TLS, другие подходы казались вполне адекватными – не потому, что теоретических слабостей не существовало, а потому, что наличие теоретических слабостей еще не означает фактической уязвимости.

Шифры с аутентификацией

Шифры с аутентификацией – альтернатива комбинации шифра и MAC. Они похожи на обычные шифры, но возвращают аутентификационный жетон наряду с шифртекстом.

Шифрование с аутентификацией описывается формулой $\mathbf{AE}(K, P) = (C, T)$. Акроним **AE** означает *authenticated encryption*, и, как мы видим, шифру передается ключ (K) и открытый текст (P), а возвращает он шифртекст (C) и сгенерированный жетон (T). Иначе говоря, один шифртекст с аутентификацией делает то же самое, что комбинация шифра и MAC, только проще, быстрее и зачастую безопаснее.

Дешифрирование шифра с аутентификацией описывается формулой $\mathbf{AD}(K, C, T) = P$. Здесь **AD** означает *authenticated decryption*; алгоритм получает шифртекст (C), жетон (T), ключ (K) и возвращает открытый текст (P). Если C или T некорректны, то **AD** возвращает ошибку, и получатель отказывается от обработки открытого текста, который может быть подделан. С другой стороны, если **AD** вернул от-

крытый текст, то есть уверенность, что он был зашифрован тем, кто знает секретный ключ.

Основные требования к шифру с аутентификацией просты: аутентификация должна быть такой же стойкой, как в случае MAC, т. е. должно быть невозможно подделать пару (*C*, *T*) так, что функция **AD** примет и дешифрирует ее.

Что касается конфиденциальности, то шифр с аутентификацией принципиально более стойкий, чем обычный, потому что системы, хранящие секретный ключ, станут дешифрировать шифртекст, только если аутентификационный жетон действителен. В противном случае шифртекст будет отброшен. Эта особенность не дает противнику провести атаку с подобранным шифртекстом, когда он создает шифртекст и просит вернуть соответствующий ему открытый текст.

Шифрование с аутентификацией и ассоциированными данными

Криптографы определяют *ассоциированные данные* как любые данные, обрабатываемые шифром с аутентификацией, которые аутентифицируются (благодаря аутентификационному жетону), но не шифруются. По умолчанию все открытые данные, подаваемые на вход шифра с аутентификацией, шифруются *и* аутентифицируются.

Но что, если мы просто хотим аутентифицировать все сообщение, включая его незашифрованные части, но не хотим шифровать его целиком? То есть мы хотим аутентифицировать и передать некоторые данные в дополнение к зашифрованному сообщению. Например, если шифр обрабатывает сетевой пакет, состоящий из заголовка и полезной нагрузки, то мы можем решить, что полезную нагрузку следует зашифровать, чтобы скрыть передаваемые данные, а заголовок шифровать не нужно, потому что он содержит информацию, необходимую для доставки пакета конечному получателю. Однако в то же время хотелось бы аутентифицировать данные заголовка – так мы будем уверены, что он получен от ожидаемого отправителя.

Для достижения этих целей было введено понятие шифрования с аутентификацией и ассоциированными данными (authenticated encryption with associated data – AEAD). Алгоритм AEAD позволяет присоединить открытые данные к шифртексту таким образом, что при повреждении открытых данных аутентификационный жетон не пройдет проверку и шифртекст не будет дешифрироваться.

Операцию AEAD можно записать в виде **AEAD**(*K*, *P*, *A*) = (*C*, *A*, *T*). Получив ключ (*K*), открытый текст (*P*) и ассоциированные данные (*A*), AEAD возвращает шифртекст, незашифрованные ассоциированные данные и аутентификационный жетон. AEAD оставляет незашифрованные ассоциированные данные без изменения, а в качестве шифртекста возвращает результат шифрования открытого текста. Аутентификационный жетон зависит от *P* и *A* и будет считаться действительным, только если ни *C*, ни *A* не были модифицированы.

Поскольку аутентификационный жетон зависит от *A*, дешифрирование с ассоциированными данными вычисляется как **ADAD**(*K*, *C*, *A*, *T*) = (*P*, *A*). Для дешифрирования необходим ключ, шифртекст, ассоциированные данные и жетон. Если *C* или *A* повреждены, дешифрирование завершается неудачно.

Отметим, что при использовании AEAD *A* или *P* можно оставить пустыми. Если ассоциированные данные *A* не заданы, то AEAD превращается в обычный шифр с аутентификацией, а если не задан *P*, то мы имеем просто MAC.

Примечание *На момент написания этой книги AEAD считается нормой для шифрования с аутентификацией. Поскольку почти все используемые в настоящее время шифры с аутентификацией поддерживают ассоциированные данные, все упоминания шифров с аутентификацией в этой книге на самом деле относятся к AEAD, если только явно не оговорено противное. Говоря об операциях шифрования и дешифрирования в AEAD, я буду обозначать их соответственно* ***AE*** *и* ***AD****.*

Предотвращение предсказуемости с помощью одноразовых чисел

Напомним (см. главу 1), что для того, чтобы считаться безопасной, схема шифрования должна быть непредсказуемой и возвращать разные шифртексты при повторных вызовах для шифрования одного и того же открытого текста; в противном случае противник сможет определить, что дважды был зашифрован один и тот же текст. Для обеспечения непредсказуемости блочным и потоковым шифрам передается дополнительный параметр: начальное значение (IV) или одноразовое число, которое можно использовать только один раз. В шифрах с аутентификацией используется такой же прием. Поэтому шифрование с аутентификацией можно выразить в виде **AE**(*K*, *P*, *A*, *N*), где *N* – одноразовое число. Ответственность за выбор одноразового числа, которое раньше никогда не использовалось с этим ключом, возлагается на операцию шифрования.

Как и в случае блочных и потоковых шифров, дешифрирование шифра с аутентификацией требует одноразового числа, использованного при шифровании. Поэтому дешифрирование можно описать формулой **AD**(*K*, *C*, *A*, *T*, *N*) = (*P*, *A*), где *N* – одноразовое число, которое было использовано при создании *C* и *T*.

Какой шифр с аутентификацией считать хорошим?

С начала 2000-х годов исследователи стараются определить, что такое хороший шифр с аутентификацией, и сейчас, когда я пишу эти строки, ответ все еще ускользает. Из-за большого количества входных данных AEAD, играющих разные роли, определить понятие безопасности труднее, чем для обычных шифров, которые только зашифровывают сооб-

щение. Тем не менее в этом разделе я попробую описать самые важные критерии, которые следует учитывать при оценке безопасности, производительности и функциональности шифра с аутентификацией.

Критерии безопасности

Самый важный критерий при измерении стойкости шифра с аутентификацией – его способность защитить конфиденциальность данных (т. е. секретность открытого текста), а также подлинность и целостность связи (как имитовставка, способная обнаруживать поврежденные сообщения). Шифр с аутентификацией должен быть конкурентоспособен в обеих категориях: конфиденциальность должна быть не хуже, чем у самого стойкого шифра, а подлинность – такой же несомненной, как у лучших MAC. Иными словами, если убрать из AEAD аутентификацию, то должен получиться безопасный шифр, а если убрать шифрование – то стойкая MAC.

Еще одна мера безопасности шифра с аутентификацией основана на чуть более тонком понятии – хрупкости перед лицом повторения одноразовых чисел. А именно если одноразовое число все же использовано повторно, сможет ли противник дешифрировать шифртексты или узнать о различии между открытыми текстами?

Исследователи называют этот аспект надежности *стойкостью к неправильному использованию* (misuse resistance) и спроектировали стойкие к неправильному использованию шифры с аутентификацией, чтобы оценить влияние повтора одноразового числа и попытаться определить, будет ли конфиденциальность, подлинность или то и другое сразу скомпрометированы при такой атаке, а также понять, какая именно информация о зашифрованных данных может утечь.

Критерии производительности

Как и для любого криптографического алгоритма, пропускную способность шифра с аутентификацией можно измерить в битах, обработанных в секунду. Эта скорость зависит от числа операций, выполняемых алгоритмом шифра, и от дополнительных затрат на аутентификацию. Понятно, что дополнительные средства безопасности обходятся не бесплатно. Однако измерение производительности шифра не сводится только к быстродействию. Речь также идет о параллелизуемости, о структуре и о возможности применения к данным, поступающим потоком. Рассмотрим эти аспекты более внимательно.

Параллелизуемость шифра – это мера его способности обрабатывать одновременно несколько блоков данных, не ожидая завершения обработки предыдущего блока. Конструкции на основе блочных шифров легко распараллеливаются, когда каждый блок можно обработать независимо от других. Например, блочный шифр, работающий в режиме CTR (см. главу 4), – параллелизуемый, а работающий в режиме CBC – нет, потому что блоки сцепляются.

Внутренняя структура шифра с аутентификацией – еще один важный критерий производительности. Есть два основных типа структу-

ры: одноуровневая и двухуровневая. В двухуровневой структуре (например, в широко распространенном шифре AES-GCM) один алгоритм обрабатывает открытый текст, а затем другой алгоритм обрабатывает результат. Как правило, на первом уровне находится шифрование, а на втором – аутентификация. Но легко понять, что двухуровневая структура усложняет реализацию и ведет к замедлению.

Шифр с аутентификацией *допускает потоковую обработку* (употребляют также термин *онлайновый*), если он может обрабатывать сообщение блок за блоком и отбрасывать уже обработанные блоки. С другой стороны, шифры, не допускающие потоковой обработки, должны хранить сообщение целиком, обычно вследствие того, что необходимо два прохода по данным: один от начала до конца, а второй – от конца к началу данных, полученных после первого прохода.

Из-за потенциально высокого потребления памяти некоторые приложения не могут работать с шифрами, не допускающими потоковой обработки. Например, маршрутизатор мог бы получить зашифрованный блок данных, дешифрировать его и вернуть блок открытого текста, перед тем как переходить к дешифрированию следующего блока, хотя получателю дешифрированного сообщения все равно нужно будет проверить аутентификационный жетон, находящийся в конце дешифрированного потока данных.

Функциональные критерии

Функциональные критерии – это свойства шифра или его реализации, не связанные напрямую с безопасностью или производительностью. Например, некоторые шифры с аутентификацией позволяют размещать ассоциированные данные только перед данными, подлежащими шифрованию (потому что их необходимо знать, чтобы приступить к шифрованию). Другие требуют, чтобы ассоциированные данные располагались после шифруемых, или поддерживают размещение ассоциированных данных в любом месте – даже между блоками открытого текста. Последняя возможность считается наилучшей, потому что позволяет пользователям защитить свои данные в любой мыслимой ситуации, но и реализовать ее безопасно труднее всего. Как всегда, чем больше возможностей, тем выше сложность, а вместе с ней и количество потенциальных уязвимостей.

Еще один куст функциональных критериев связан с тем, можно ли использовать один и тот же базовый алгоритм и для шифрования, и для дешифрирования. Например, многие шифры с аутентификацией основаны на блочном шифре AES, в котором для шифрования и дешифрирования блока применяются похожие алгоритмы. В главе 4 мы обсуждали, что блочный шифр в режиме CBC нуждается в обоих алгоритмах, а в режиме CTR – только в алгоритме шифрования. Так и шифры с аутентификацией необязательно нуждаются в обоих алгоритмах. Хотя дополнительные затраты на реализацию двух алгоритмов – шифрования и дешифрирования – обычно несущественны при программной реализации, они часто влияют на стоимость специали-

зированного оборудования низкой ценовой категории, цена которого зависит от количества логических элементов или площади кристалла, занятой криптографическими схемами.

AES-GCM: стандартный шифр с аутентификацией

AES-GCM – самый распространенный шифр с аутентификацией. Конечно, он основан на алгоритме AES, а режим счетчика с аутентификацией Галуа (GCM) – это по существу модификация режима CTR, включающая небольшую эффективную компоненту для вычисления аутентификационного жетона. На момент написания книги AES-GCM был единственным шифром с аутентификацией, стандартизованным NIST (SP 800-38D). AES-GCM также является частью Комплекта В АНБ и безопасных сетевых протоколов IPSec, SSH и TLS 1.2, специфицированных Инженерным советом интернета (IETF).

Примечание *Хотя режим GCM работает для любого блочного шифра, скорее всего, вы встретите его только в комбинации с AES. Некоторые не хотят использовать AES, потому что это американский шифр, и по той же причине не пользуются режимом GCM. Так что GCM редко употребляется в связке с другими шифрами.*

Внутреннее устройство GCM: CTR и GHASH

На рис. 8.2 показано, как работает AES-GCM: экземпляры AES, параметризованные секретным ключом (*K*), преобразуют блок, состоящий из одноразового числа (*N*), конкатенированного со счетчиком (в данном случае начинается с 1, затем увеличивается до 2, 3 и т. д.), после чего к результату и блоку открытого текста применяется XOR для получения блока шифртекста. До сих пор ничего нового по сравнению с режимом CTR.

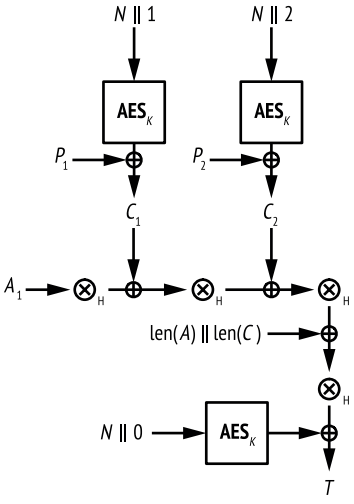

Рис. 8.2. Режим AES-GCM, применяемый к одному ассоциированному блоку данных, A_1, и двум блокам открытого текста, P_1 и P_2. Знак умножения в кружочке представляет полиномиальное умножение на H, ключ аутентификации, сформированный по K

Затем блоки шифртекста перемешиваются с применением комбинации операций XOR и умножения (как будет описано ниже). Как видим, AES-GCM выполняет 1) шифрование в режиме CTR и 2) вычисление MAC по блокам шифртекста. Поэтому AES-GCM принадлежит к классу шифрование-затем-MAC, где AES-CTR шифрует с помощью 128-битового ключа (*K*) и 96-битового одноразового ключа (*N*), а мелкое отличие заключается в том, что счетчик начинается с 1, а не с 0, как в обычном режиме CTR (с точки зрения безопасности, это не играет роли).

Для аутентификации шифртекста в GCM используется имитовставка Вегмана–Картера (см. главу 7), которая с помощью XOR объединяет значение **AES**(*K*, *N*||0) с выходом универсальной функции хеширования *GHASH*. На рис. 8.2 функции GHASH соответствует серия операций $\oplus_H$, сопровождаемая XOR с len(*A*)||len(*C*), т. е. длиной *A* (ассоциированных данных) в битах, конкатенированной с длиной *C* (шифртекста) в битах.

Таким образом, мы можем выразить значение аутентификационного жетона в виде $T = \mathbf{GHASH}(H, C) \oplus \mathbf{AES}(K, N||0)$, где *C* – шифртекст, а *H* – *хеш-ключ*, или *ключ аутентификации*. Этот ключ определен как $H = \mathbf{AES}(K, 0)$, т. е. результат шифрования блока, состоящего из последовательности нулевых байтов (этот шаг опущен на рис. 8.2 для ясности).

Примечание *В режиме GCM функция GHASH не использует ключ K напрямую. Это гарантирует, что если ключ GHASH окажется скомпрометирован, то главный ключ K останется секретным. Зная K, можно получить H, вычислив* **AES**(*K*, 0)*, но восстановить K по этому значению нельзя, потому что K здесь выступает в роли ключа AES.*

Как показано на рис. 8.2, в GHASH используется *полиномиальная нотация* для описания умножения каждого блока шифртекста на ключ аутентификации *H*. Благодаря такому использованию полиномиальной нотации GHASH работает быстро как при аппаратной, так и при программной реализации, поскольку во многих микропроцессорах имеется специальная команда полиномиального умножения (`CLMUL`, или carry-less multiplication, умножение без переносов).

К сожалению, GHASH далека от идеала. Во-первых, ее быстродействие не оптимально. Даже при использовании команды `CLMUL` уровень AES-CTR, который зашифровывает открытый текст, все равно быстрее, чем GHASH MAC. Во-вторых, GHASH очень трудно реализовать корректно. Даже опытные разработчики проекта OpenSSL, самого широко используемого криптографического ПО, умудрились реализовать функцию GHASH для AES-GCM неправильно. В одной из фиксаций была функция `gcm_ghash_clmul`, которая позволяла противнику подделать действительные имитовставки для AES-GCM. (К счастью, эту ошибку обнаружили инженеры Intel до того, как она попала в очередную версию OpenSSL.)

Полиномиальное умножение

Для нас полиномиальное умножение, очевидно, сложнее, чем классическое арифметическое умножение, но для компьютеров оно проще, потому что отсутствуют переносы. Например, пусть требуется вычислить произведение полиномов $(1 + X + X^2)$ и $(X + X^3)$. Сначала умножаем оба полинома, как обычно (два члена X^3 взаимно уничтожаются):

$$(1 + X + X^2) \otimes (X + X^3) = X + X^3 + X^2 + X^4 + X^3 + X^5$$
$$= X + X^2 + X^4 + X^5.$$

Теперь применяем сокращение по модулю: $X + X^2 + X^4 + X^5$ по модулю $1 + X^3 + X^4$ дает X^2, потому что $X + X^2 + X^4 + X^5$ можно записать в виде $X + X^2 + X^4 + X^5 = X \otimes (1 + X^3 + X^4) + X^2$. Вообще, $A + BC$ по модулю B равно A, в силу определения сокращения по модулю.

Безопасность GCM

Самая серьезная слабость AES-GCM – уязвимость к повторению одноразового числа. Если одно и то же одноразовое число N используется дважды, то противник сможет получить ключ аутентификации H и с его помощью подделывать жетоны для любого шифртекста, ассоциированных данных или их комбинации.

Ознакомление с простой алгеброй, стоящей за вычислениями AES-GCM (показанными на рис. 8.2), поможет понять причины этой уязвимости. Жетон (T) вычисляется в виде $T = \mathbf{GHASH}(H, A, C) \oplus \mathbf{AES}(K, N||0)$, где GHASH – универсальная хеш-функция с линейной связью между входами и выходами.

И что будет, если вычислить два жетона, T_1 и T_2, с одним и тем же одноразовым числом N? Правильно, члены AES пропадут. Если $T_1 = \mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{AES}(K, N||0)$ и $T_2 = \mathbf{GHASH}(H, A_2, C_2) \oplus \mathbf{AES}(K, N||0)$, то, применив к ним операцию XOR, получим:

$$\mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{AES}(K, N||0) \oplus \mathbf{GHASH}(H, A_2, C_2) \oplus \mathbf{AES}(K, N||0)$$
$$= \mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{GHASH}(H, A_2, C_2) \oplus \mathbf{AES}(K, N||0) \oplus \mathbf{AES}(K, N||0)$$
$$= \mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{GHASH}(H, A_2, C_2).$$

Итак, если одно и то же одноразовое число используется дважды, то противник может восстановить значение $\mathbf{GHASH}(H, A_1, C_1) \oplus \mathbf{GHASH}(H, A_2, C_2)$ для некоторых известных A_1, C_1, A_2 и C_2. Затем, в силу линейности GHASH, противник сможет легко определить H. (Было бы еще хуже, если бы в GHASH использовался тот же ключ K, что и при шифровании, но, поскольку $H = \mathbf{AES}(K, 0)$, простого способа определить K по H не существует.)

Совсем недавно – в 2016 году – исследователи просканировали интернет на предмет экземпляров AES-GCM в HTTPS-серверах; их ин-

тересовали системы с повторяющимися одноразовыми числами (см. *https://eprint.iacr.org/2016/475/*). Обнаружилось 184 таких сервера, из них 23 всегда использовали в качестве одноразового числа строку из одних нулей.

Эффективность GCM

Одно из преимуществ режима GCM заключается в том, что шифрование и дешифрирование допускают распараллеливание, т. е. шифровать и дешифрировать различные блоки открытого текста можно независимо. Однако вычисление MAC в AES-GCM не распараллеливается, потому что его нужно вычислять по всему шифртексту, после того как GHASH обработала все ассоциированные данные. Это означает, что система, получившая сначала открытый текст, а потом ассоциированные данные, должна будет подождать, пока все ассоциированные данные будут прочитаны и хешированы, и только потом сможет приступить к хешированию первого блока шифртекста.

Тем не менее GCM допускает потоковую обработку: поскольку вычисления, производимые обоими уровнями, можно организовать в виде конвейера, нет необходимости хранить все блоки шифртекста перед вычислением GHASH, потому что GHASH будет обрабатывать каждый блок сразу после его зашифровки. Иными словами, P_1 зашифровывается в C_1, затем GHASH обрабатывает C_1, а в это время P_2 зашифровывается в C_2, после чего P_1 и C_1 уже не нужны. И так далее.

OCB: шифр с аутентификацией, более быстрый, чем GCM

Акроним *OCB* означает «offset codebook» (*кодовая книга со смещением*) (хотя его автор, Фил Рогуэй, предпочитает называть его просто OCB). Разработанный в 2001 году, OCB появился раньше GCM и, как и GCM, порождает шифр с аутентификацией по блочному шифру, но при этом он более простой и быстрый. Тогда почему же OCB не обрел популярность? К сожалению, до 2013 года для использования OCB требовалось получать лицензию от автора. Но теперь Рогуэй выдает бесплатные лицензии для реализаций невоенного программного обеспечения (см. *http://web.cs.ucdavis.edu/~rogaway/ocb/license.htm*). Поэтому, хотя OCB пока формально не стандартизован, вероятно, он получит более широкое распространение.

В отличие от GCM, OCB смешивает шифрование и аутентификацию в одном уровне обработки с единственным ключом. Отдельный компонент аутентификации отсутствует, поэтому OCB работает почти так же быстро, как шифры без аутентификации, а аутентификация обходится почти бесплатно. На самом деле режим OCB почти такой же простой, как ECB (см. главу 4), но при этом безопасный.

Внутреннее устройство OCB

На рис. 8.3 показано, как работает OCB: каждый блок открытого текста P преобразуется в блок шифртекста $C = \mathbf{E}(K, P \oplus O) \oplus O$, где **E** – функция шифрования блочного шифра. Здесь O (*смещение*) – значение, зависящее от ключа и одноразового числа, которое увеличивается при обработке каждого нового блока.

Для порождения аутентификационного жетона OCB сначала вычисляет XOR всех блоков открытого текста, $S = P_1 \oplus P_2 \oplus P_3 \oplus \ldots$. Тогда жетон $T = \mathbf{E}(K, S \oplus O^*)$, где O^* – смещение, вычисленное по смещению последнего обработанного блока открытого текста.

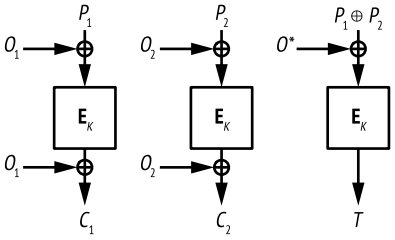

Рис. 8.3. Процесс шифрования в OCB при наличии двух блоков открытого текста без ассоциированных данных

Как и AES-GCM, OCB поддерживает ассоциированные данные в виде последовательности блоков $A_1, A_2 \ldots$. Если зашифрованное OCB сообщение содержит ассоциированные данные, то аутентификационный жетон вычисляется по формуле

$$T = \mathbf{E}(K, S \oplus O^*) \oplus \mathbf{E}(K, A_1 \oplus O_1) \oplus \mathbf{E}(K, A_2 \oplus O_2) \oplus \ldots,$$

где OCB определяет смещения, отличные от тех, что использовались при шифровании P.

В отличие от GCM и схемы шифрование-затем-MAC, где аутентификационный жетон создается по шифртексту, в OCB жетон вычисляется по данным открытого текста. В этом нет ничего плохого, и безопасность OCB подкреплена солидными доказательствами.

Примечание *Дополнительные сведения о том, как корректно реализовать OCB, см. либо в RFC 7253, либо в статье Krovetz and Rogaway «The Software Performance of Authenticated-Encryption Modes» 2011 года, в которой рассматривается последняя и лучшая версия OCB – OCB3. Также см. FAQ, посвященный OCB, по адресу http://web.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm.*

Безопасность OCB

OCB чуть менее уязвим к повторению одноразовых чисел, чем GCM. Так, если одноразовое число использовано дважды, то противник, который видит оба шифртекста, заметит, что, скажем, третий блок открытого текста первого сообщения идентичен третьему блоку от-

крытого текста второго сообщения. В случае GCM противник может найти не только дубликаты, но и XOR-разность блоков в одной и той же позиции. Поэтому повторение одноразовых чисел в случае GCM оказывается хуже, чем в случае OCB.

Как и в GCM, повторные одноразовые числа могут нарушить аутентичность OCB, хотя и не так ужасно. Например, противник мог бы скомбинировать блоки двух сообщений, аутентифицированных с помощью OCB, и создать еще одно зашифрованное сообщение с такой же контрольной суммой и жетоном, как одно из двух оригинальных сообщений, но восстановить секретный ключ, как в GCM, он не сможет.

Эффективность OCB

Скорость работы OCB и GCM почти одинакова. Как и GCM, OCB допускает распараллеливание и потоковую обработку данных. Если говорить о «голой» эффективности, то GCM и OCB совершают примерно одинаковое количество обращений к базовому блочному шифру (обычно AES), но OCB немного эффективнее, потому что просто выполняет операции XOR с открытым текстом, а не относительно дорогое вычисление GHASH. (В ранних поколениях микропроцессоров Intel AES-GCM работал в три и более раз медленнее AES-OCB, потому что команды AES и GHASH вынуждены были конкурировать за ресурсы CPU и не могли выполняться параллельно.)

Важное различие между реализациями OCB и GCM заключается в том, что OCB нужны функции шифрования и дешифрирования блочного шифра, что увеличивает стоимость аппаратных реализаций, если для криптографических компонентов доступна ограниченная площадь на кристалле. Напротив, в GCM для шифрования и дешифрирования используется только функция шифрования.

SIV: самый безопасный шифр с аутентификацией?

Synthetic IV, или SIV, – режим шифра с аутентификацией, обычно используемый совместно с AES. В отличие от GCM и OCB, SIV безопасен, даже если одно и то же одноразовое число используется дважды: получив два шифртекста, зашифрованных с одним одноразовым числом, противник не сможет узнать, был ли в обоих случаях зашифрован один и тот же открытый текст. В отличие от сообщений, зашифрованных GCM или OCB, противник не сможет сказать, одинаковы ли первые блоки обоих сообщений, потому что применяемое при шифровании одноразовое число сначала вычисляется в виде комбинации заданного одноразового числа и открытого текста.

Спецификация построения SIV более общая, чем для GCM. Вместо детального описания внутреннего устройства функции GHASH, применяемой в GCM, спецификация просто сообщает, как скомбинировать шифр (**E**) и псевдослучайную функцию (**PRF**) для получения

шифра с аутентификацией. Именно: следует вычислить жетон T = **PRF**(K_1, $N||P$), а затем шифртекст C = **E**(K_2, T, P), где T играет роль одноразового числа для **E**. Таким образом, SIV нуждается в двух ключах (K_1 и K_2) и одноразовом числе (N).

Главная проблема SIV – то, что он не допускает потоковой обработки: после вычисления T необходимо хранить весь открытый текст P в памяти. Иными словами, чтобы зашифровать открытый текст размером 100 ГБ с помощью SIV, нужно сначала сохранить все 100 ГБ, чтобы функция шифрования SIV могла их прочитать.

В документе RFC 5297, основанном на статье Rogaway and Shrimpton «Deterministic Authenticated-Encryption» 2006 года, сказано, что в SIV используется CMAC-AES (построение MAC с помощью AES) в качестве PRF и AES-CTR в качестве шифра. В 2015 году была предложена более эффективная версия SIV, GCM-SIV, в которой быстрая хеш-функция GHASH из GCM комбинируется с режимом SIV, в результате чего получается почти такой же быстрый шифр, как GCM. Но, как и оригинальный SIV, версия GCM-SIV не допускает потоковой обработки. (Дополнительные сведения см. по адресу *https://eprint.iacr.org/2015/102/*.)

AEAD на основе перестановки

Теперь поговорим о совершенно другом подходе к построению шифра с аутентификацией: вместо нового режима работы блочного шифра, например AES, мы рассмотрим шифр на основе перестановки. Перестановка – это просто обратимое преобразование входа в выход такого же размера, для которого не нужен ключ; проще не придумаешь. И вдобавок получающийся AEAD быстрый, доказуемо безопасный и более стойкий к повторному использованию одноразового числа, чем GCM и OCB.

На рис. 8.4 показано, как работает AEAD на основе перестановки: отправляясь от фиксированного начального состояния H_0, мы применяем XOR к конкатенации ключа K с одноразовым числом N и внутреннему состоянию. В результате получается новое внутреннее состояние того же размера, что первоначальное. Затем новое состояние подвергается перестановке **P**. Теперь первый блок открытого текста P_1 объединяется операцией XOR с текущим состоянием, и результат принимается в качестве первого блока шифртекста. Предполагается, что размер P_1 и C_1 одинаков, но меньше размера состояния.

Чтобы зашифровать второй блок, мы применяем к состоянию перестановку **P** и выполняем XOR следующего блока простого текста P_2 с текущим состоянием – результат и будет значением C_2. Эта процедура повторяется для всех блоков открытого текста, а после последнего применения **P** биты внутреннего состояния становятся аутентификационным жетоном T, как показано справа на рис. 8.4.

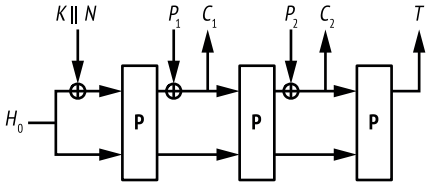

Рис. 8.4. Шифр с аутентификацией на основе перестановки

Примечание *Режим, показанный на рис. 8.4, можно адаптировать для поддержки ассоциированных данных, но этот процесс довольно сложный, поэтому его описание мы опустим.*

При проектировании шифров с аутентификацией на основе перестановки необходимо соблюсти некоторые требования, обеспечивающие безопасность. Во-первых, отметим, что с помощью XOR мы объединяем входные значения только с частью состояния: чем больше эта часть, тем сильнее контроль противника над внутренним состоянием в случае успешной атаки, а значит, тем ниже безопасность шифра. В самом деле, безопасность целиком зависит от секретности внутреннего состояния.

Кроме того, блоки нужно надлежащим образом дополнять битами, так чтобы любые два разных сообщения давали разные результаты. Так, если последний блок открытого текста короче полного блока, то его не следует дополнять нулями, в противном случае двухбайтовый блок открытого текста 0000 был бы дополнен до полного блока 0000…0000, как и трехбайтовый блок 000000. В результате жетоны обоих сообщений совпали бы, хотя их размер разный.

А что, если одноразовое число используется в шифре на основе перестановки несколько раз? Хорошая новость заключается в том, что последствия не так серьезны, как в случае GCM или OCB, – стойкость аутентификационного жетона не будет скомпрометирована. Если одноразовое число повторяется, то противник сможет узнать только, что два зашифрованных сообщения начинаются одинаковым префиксом, а также длину этого префикса. Например, хотя шифрование двух шестиблочных сообщений *ABCXYZ* и *ABCDYZ* (здесь каждая буква соответствует блоку) с одним и тем же одноразовым числом может дать два шифртекста *JKLTUV* и *JKLMNO* с одинаковыми префиксами, противник не сможет узнать, что последние два блока обоих открытых текстов совпадают (*YZ*).

Что касается производительности, то у шифров на основе перестановок имеются такие преимущества: один уровень операций, возможность потоковой обработки и использование одного базового алгоритма для шифрования и дешифрирования. Однако они не допускают распараллеливания, как GCM или OCB, потому что следующее обращение к **P** возможно только после завершения предыдущего.

Примечание *Если вы испытываете искушение взять свою любимую перестановку и на ее основе создать собственный шифр с аутентификацией, не делайте этого. Скорее всего, вы напортачите в де-*

талях и получите небезопасный шифр. Почитайте написанные опытными криптографами спецификации таких алгоритмов, как Keyak (производный от Keccak) и NORX (спроектированный Филиппом Йовановичем, Сэмюэлем Нивсом и мной), и вы поймете, что шифры на основе перестановок гораздо сложнее, чем может показаться на первый взгляд.

Какие возможны проблемы

Поверхность атаки на шифры с аутентификацией больше, чем на хеш-функции или блочные шифры, потому что их задача – обеспечить конфиденциальность *и* подлинность сообщения. Они принимают различные входные значения и должны оставаться безопасными вне зависимости от входов, будь то только ассоциированные данные и никаких зашифрованных, очень большие открытые тексты или ключи разного размера. Они также должны сохранять безопасность при всех значениях одноразовых чисел и противостоять атакам со сбором многочисленных пар сообщение–жетон, в том числе (до некоторой степени) при случайном повторении одноразовых чисел.

Таким требованиям нелегко удовлетворить, и, как мы увидим далее, даже у шифра AES-GCM есть несколько недостатков.

AES-GCM и слабые хеш-ключи

Одна из слабостей AES-GCM связана с его алгоритмом аутентификации GHASH: некоторые значения хеш-ключа H существенно упрощают атаки против механизма аутентификации GCM. Точнее, если значение H принадлежит строго математически определенным подгруппам множества всех 128-битовых строк, то противник может угадать правильный аутентификационный жетон для некоторого сообщения, просто перетасовав блоки предыдущего сообщения.

Чтобы лучше разобраться в этом, посмотрим, как работает GHASH.

Внутренний механизм GHASH

На рис. 8.2 мы видели, что GHASH начинает работу со 128-битового значения H, которое первоначально равно $\mathbf{AES}(K, 0)$, а затем итеративно вычисляет выражение

$$X_i = (X_{i-1} \oplus C_i) \otimes H,$$

начиная с $X_0 = 0$ и обрабатывая блоки шифртекста $C_1, C_2, \ldots$. Последнее значение X_i возвращается для вычисления окончательного жетона.

Теперь предположим для простоты, что все значения C_i равны 1, так что для любого i имеем:

$$C_i \otimes H = 1 \otimes H = H.$$

Тогда из формулы GHASH

$$X_i = (X_{i-1} \oplus C_i) \otimes H$$

получаем

$$X_1 = (X_0 \oplus C_1) \otimes H = (0 \oplus 1) \otimes H = H,$$

поскольку при подстановке 0 вместо X_0 и 1 вместо C_1 имеем:

$$(0 \oplus 1) = 1.$$

Пользуясь свойством дистрибутивности $\otimes$ относительно $\oplus$, подставляем H вместо X и 1 вместо C_2 и вычисляем следующее значение X_2

$$X_2 = (X_1 \oplus C_2) \otimes H = (H \oplus 1) \otimes H = H^2 \oplus H,$$

где H^2 по определению равно $H \otimes H$.

Теперь выведем X_3:

$$X_3 = (X_2 \oplus C_3) \otimes H = (H^2 \oplus H \oplus 1) \otimes H = H^3 \oplus H^2 \oplus H.$$

Далее мы находим, что $X^4 = H^4 \oplus H^3 \oplus H^2 \oplus H$, и так далее. В итоге получается, что последнее X_i равно:

$$X_n = H^n \oplus H^{n-1} \oplus H^{n-2} \oplus \ldots H^2 \oplus H.$$

Вспомним, что мы предположили, что все блоки C_i равны 1. Если бы они могли принимать произвольные значения, то вместо этой формулы мы получили бы такую:

$$X_n = C_1 \otimes H^n \oplus C_2 \otimes H^{n-1} \oplus C_3 \otimes H^{n-2} \oplus \ldots \oplus C_{n-1} \otimes H^2 \oplus C_n \otimes H.$$

Затем GHASH выполнила бы XOR этого последнего X_n с длиной сообщения, умножила бы результат на H и объединила бы через XOR с $\mathbf{AES}(K, N||0)$ – это и был бы аутентификационный жетон T.

Где собака зарыта?

И что же здесь может пойти не так? Рассмотрим сначала два простейших случая:

- если $H = 0$, то $X_n = 0$ вне зависимости от значений C_i, а значит, и вне зависимости от сообщения. Следовательно, при H, равном 0, у всех сообщений будет одинаковый жетон;
- если $H = 1$, то жетон равен результату применения XOR ко всем блокам шифртекста, и при изменении порядка этих блоков жетон не изменится.

Разумеется, 0 и 1 – лишь два из 2^{128} возможных значений H, и встретиться они могут с вероятностью $2/2^{128} = 1/2^{127}$. Но есть и другие слабые значения, а именно все значения H, принадлежащие *короткому циклу* при возведении в степени. Например, значение H = 10d04d25f93556e69f58ce2f8d035a4 принадлежит циклу длины 5, потому что для него $H^5 = H$ и, значит, $H^e = H$ для любого e, кратного пяти. Следовательно, в приведенном выше выражении последнего значения GHASH X_n перестановка блоков Cn (умножаемого на H) и C_{n-4} (умножаемого на H^5) оставит аутентификационный жетон неизменным, что эквивалентно подделке. Противник может воспользоваться этим свойством для конструирования нового сообщения вместе с правильным жетоном, не зная ключа, – для безопасного шифра с аутентификацией такое должно быть невозможно.

Описанный выше пример основан на цикле длины 5, но есть много циклов большей длины, а значит, много значений H, которые слабее, чем должны быть. Вывод: в том маловероятном случае, когда H принадлежит короткому циклу, противник может подделать столько жетонов, сколько захочет, но, не зная H или K, он не сможет определить длину цикла H. Таким образом, хотя эту уязвимость невозможно эксплуатировать, ее все же следует избегать, более тщательно выбирая полином, используемый для сокращения по модулю.

Примечание *Дополнительные сведения об этой атаке можно почерпнуть из статьи Markku-Juhani O. Saarinen «Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes», доступной по адресу https://eprint.iacr.org/2011/202/.*

AES-GCM и короткие жетоны

На практике AES-GCM обычно возвращает 128-битовые жетоны, но может порождать жетоны любой длины. К сожалению, при использовании коротких жетонов вероятность подделки значительно возрастает.

Если длина жетона равна 128 бит, то противник, пытающийся подделать жетон, добьется успеха с вероятностью $1/2^{128}$, потому что существует 2^{128} возможных жетонов. (В общем случае для n-битовых жетонов вероятность успеха должна быть равна $1/2^n$.) Но при использовании коротких жетонов вероятность успешной подделки гораздо выше, чем $1/2^n$, из-за слабостей в структуре GCM, обсуждение которых выходит за рамки этой книги. Например, при 32-битовых жетонах противник, знающий аутентификационный жетон некоторого сообщения длиной 2 МБ, сможет добиться успеха с вероятностью $1/2^{16}$ вместо $1/2^{32}$.

Вообще, при n-битовых жетонах вероятность успешной подделки равна не $1/2^n$, а $2^m/2^n$, где 2^m – число блоков самого длинного сообщения, для которого противник наблюдал жетон. Например, если используются 48-битовые жетоны и обрабатываются сообщения длиной 4 ГБ (или 2^{28} блоков по 16 байт), то вероятность подделки будет

равна $2^{28}/2^{48} = 1/2^{20}$, т. е. примерно один шанс на миллион. С точки зрения криптографии, это довольно высокая вероятность. (Дополнительные сведения об этой атаке см. в статье Niels Ferguson «Authentication Weaknesses in GCM» 2005 года.)

Для дополнительного чтения

Если вы хотите узнать больше о шифрах с аутентификацией, то зайдите на домашнюю страницу конкурса CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) (*http://competitions.cr.yp.to/caesar.html*). Запущенный в 2012 году, CAESAR стал конкурсом для криптографов в духе конкурсов на звание AES и SHA-3, хотя организовывал его не NIST.

Конкурс CAESAR привлек впечатляющее количество новаторских проектов: от похожих на OCB режимов до шифров на основе перестановок, а также совсем новых базовых алгоритмов. В качестве примеров приведем уже упомянутые выше шифры с аутентификацией на основе перестановок NORX и Keyak; стойкий к неправильному использованию AEZ, построенный на основе двухуровневого режима, не допускающего потоковой обработки; AEGIS, элегантный и простой шифр с аутентификацией, в котором используется функция раунда из AES.

В этой главе мы рассматривали в основном режим GCM, но есть ряд других режимов, которые применяются в реальных приложениях. А именно режим счетчика с CBC-MAC (CCM) и режим EAX были конкурентами GCM при стандартизации в начале 2000-х годов, и хотя в итоге был выбран GCM, эти два режима встречаются в нескольких приложениях. Например, CCM применяется в протоколе WPA2 шифрования в сетях Wi-Fi. При желании можете прочитать спецификации этих шифров и обзоры их безопасности и производительности.

На этом заканчивается наше обсуждение криптографии с симметричным ключом! Мы рассмотрели блочные шифры, потоковые шифры, функции хеширования (с ключом и без ключа), а теперь еще и шифры с аутентификацией, т. е. все основные криптографические компоненты, работающие с симметричным ключом или вообще без ключа. Прежде чем перейти к криптографии с *асимметричным* ключом, мы в главе 9 остановимся на некоторых вопросах информатики и математики – это станет теоретической основой для обсуждения таких асимметричных схем, как алгоритмы RSA (глава 10) и Диффи–Хеллмана (глава 11).

9

ТРУДНЫЕ ЗАДАЧИ

Трудные вычислительные задачи – краеугольный камень современной криптографии. Такие задачи легко сформулировать, но практически невозможно решить. Это задачи, для которых даже самый лучший алгоритм найдет решение не раньше, чем погаснет Солнце.

В 1970-е годы строгое изучение трудных задач породило новую науку – теорию вычислительной сложности, которая оказала огромное влияние на криптографию и многие другие области знания, включая экономику, физику и биологию. В этой главе я опишу важные понятия и средства теории сложности, необходимые для понимания оснований криптографической безопасности, и познакомлю вас с трудными задачами, стоящими за такими схемами с открытым ключом, как алгоритм шифрования RSA и протокол совместной выработки ключа Диффи–Хеллмана. Мы затронем ряд глубоких идей, но лишь поверхностно – я постараюсь свести к минимуму технические детали. Но все же надеюсь, что вы оцените, с какой элегантностью криптография пользуется теорией вычислительной сложности для повышения безопасности.

Вычислительная трудность

Вычислительная задача – это вопрос, на который можно получить ответ, проделав достаточное количество вычислений, например «является ли 2017 простым числом?» или «сколько букв *i* в слове *incomprehensibilities*?». *Вычислительная трудность* – это общее свойство вычислительных задач, для которых не существует алгоритма с разумным временем работы. Такие задачи также называют *неразрешимыми*, поскольку на практике их зачастую невозможно решить.

Удивительно, что вычислительная трудность не зависит от типа используемого вычислительного устройства, будь то процессор общего назначения, интегрированная схема или механическая машина Тьюринга. Действительно, одно из первых открытий теории вычислительной сложности заключается в том, что все модели вычислений эквивалентны. Если задачу можно эффективно решить с помощью одного вычислительного устройства, то ее можно эффективно решить и на любом другом устройстве, перенеся алгоритм на язык этого устройства. Исключение составляют квантовые компьютеры, но их пока не существует. Таким образом, при обсуждении вычислительной трудности необязательно указывать конкретное вычислительное устройство или оборудование, достаточно обсуждать только алгоритмы.

Для оценки трудности мы сначала определим способ измерения сложности алгоритма, или время его работы. А затем по времени работы будем классифицировать задачи как трудные или легкие.

Измерение времени работы

Большинство разработчиков знакомы с *вычислительной сложностью*, или приблизительным количеством выполняемых алгоритмом операций, выраженным в виде функции от размера входных данных. Размер подсчитывается битах или в элементах ввода. Например, рассмотрим алгоритм в листинге 9.1, написанный на псевдокоде. Он ищет значение *x* в массиве *n* элементов и возвращает индекс найденной позиции.

Листинг 9.1. Простой алгоритм поиска, написанный на псевдокоде, вычислительная сложность которого линейно зависит от длины массива n. Алгоритм возвращает индекс позиции (число от 1 до n), в которой найдено значение x, или 0, если x отсутствует в массиве

```
search(x, array, n):
    for i from 1 to n {
        if (array[i] == x) {
            return i;
        }
    }
    return 0;
}
```

В этом алгоритме для поиска значения *x* применяется цикл `for`, в котором перебираются все элементы массива. На каждой итерации переменной *i* присваивается число, начиная с 1. Затем проверяется, равно ли *x* значение в *i*-й позиции массива `array`. Если да, то возвращается *i*. В противном случае мы увеличиваем *i* на единицу и проверяем следующую позицию – и так далее. Если, проверив все *n* элементов массива, мы так и не найдем *x*, то вернем значение 0.

Для алгоритмов такого типа сложность определяется как количество итераций цикла `for`: 1 в лучшем случае (если *x* равно `array[1]`), *n* в худшем случае (если *x* равно `array[`*n*`]` или *x* вообще не найдено в массиве `array`) и *n*/2 в среднем, если *x* случайно распределено среди *n* позиций массива. Если взять массив, в 10 раз больший, то алгоритм будет работать в 10 раз медленнее. Поэтому сложность пропорциональна *n*, или «линейно зависит» от *n*. Алгоритм со сложностью, линейно зависящей от *n*, считается быстрым, в отличие от алгоритма, сложность которого зависит от *n* экспоненциально. Хотя обработка большего объема данных замедляется, на практике разница обычно составляет всего несколько секунд.

Но большинство полезных алгоритмов работает медленнее, и их сложность больше линейной. Классический пример – алгоритм сортировки: если дан список *n* значений в случайном порядке, то в среднем для его сортировки понадобится $n \times \log n$ простых операций, такую сложность иногда называют *линейно-логарифмической*. Поскольку $n \times \log n$ растет быстрее *n*, скорость сортировки падает не пропорционально *n*, а быстрее. И все же такие алгоритмы сортировки остаются в области *практических* вычислений, или вычислений, которые можно выполнить за разумное время.

В какой-то момент мы достигаем потолка, практически осуществимого даже при сравнительно малой длине входных данных. Возьмем простейший пример из области криптоанализа: поиск секретного ключа полным перебором. Напомним (см. главу 1), что для заданного открытого текста *P* и шифртекста $C = \mathbf{E}(K, P)$ требуется примерно 2^n попыток для определения *n*-битового симметричного ключа, потому что всего существует 2^n возможных ключей. Это пример сложности, растущей экспоненциально. С точки зрения теории сложности, *экспоненциальная сложность* означает, что задачу практически невозможно решить, потому что при возрастании *n* объем вычислений очень быстро становится неподъемным.

Можно возразить, что мы здесь сравниваем апельсины с яблоками: в функции `search()` в листинге 9.1 подсчитывалось количество операций `if (array[i] == x)`, а при восстановлении ключа подсчитывается количество шифрований, каждое из которых в тысячи раз медленнее одного сравнения на равенство. Такая несогласованность может быть существенной при сравнении алгоритмов с очень похожей сложностью, но в большинстве случаев она не играет роли, потому что количество операций важнее стоимости одной операции. Кроме того, при оценке сложности *постоянные множители* игнорируются: говоря,

что временная сложность алгоритма составляет порядка n^3 операций (*кубическая сложность*), мы подразумеваем, что он может фактически выполнять $41 \times n^3$ или $12\,345 \times n^3$ операций – это все равно, потому что при возрастании n значимость постоянных множителей снижается до такой степени, что можно их вообще игнорировать. Предметом анализа сложности является теоретическая трудность как функция от размера входных данных; точное количество процессорных тактов на конкретном компьютере не имеет значения.

Для выражения сложности часто применяется нотация $O()$ («О большое»). Например, $O(n^3)$ означает, что сложность растет не быстрее, чем n^3, без учета постоянных множителей. Нотация $O(1)$ означает, что для работы алгоритма требуется *постоянное время*, не зависящее от длины входных данных! Например, алгоритм, который определяет четность целого числа по его младшему биту и возвращает «четное», если этот бит равен 0, и «нечетное» в противном случае, будет тратить одинаковое время вне зависимости от длины целого числа.

Для иллюстрации различий между линейной $O(n)$, квадратичной $O(n^2)$ и экспоненциальной $O(2^n)$ временными сложностями взгляните на рис. 9.1.

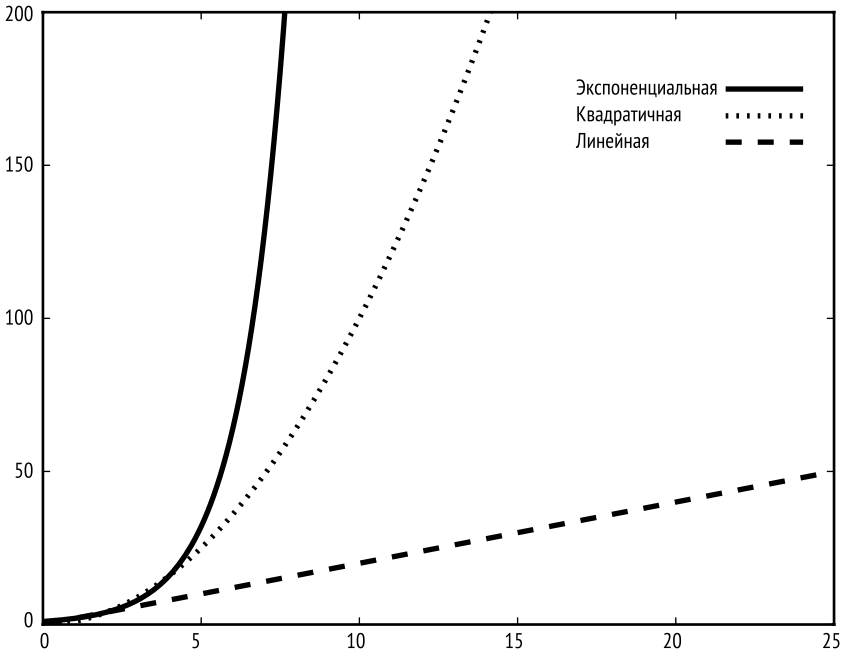

Рис. 9.1. Экспоненциальная, квадратичная и линейная сложности в порядке убывания скорости роста слева направо

Экспоненциальная сложность означает, что задача практически неразрешима, линейная – что решение возможно, а квадратичная занимает место посередине.

Полиномиальное и суперполиномиальное время

Сложность $O(n^2)$, упомянутая в предыдущем разделе (средняя кривая на рис. 9.1), – частный случай более широкого класса полиномиальной сложности $O(n^k)$, где k – фиксированное число, например 3, 2.373, 7/10 или квадратный корень из 17. Алгоритмы с полиномиальной сложностью особенно важны в теории сложности в криптографии, потому что определяют предел практически осуществимого. Алгоритм, работающий за *полиномиальное время*, завершается в разумные сроки, даже если размер входных данных велик. Поэтому для специалистов по теории сложности и криптографов полиномиальное время – синоним эффективности.

С другой стороны, алгоритмы, требующие *суперполиномиального времени*, т. е. времени $O(f(n))$, где $f(n)$ – произвольная функция, растущая быстрее любого полинома, считаются практически неосуществимыми. Я говорю «суперполиномиальная», а не просто «экспоненциальная», потому что существуют сложности, находящиеся между полиномиальной и хорошо известной экспоненциальной сложностью $O(2^n)$, например $O(n^{\log(n)})$, показанная на рис. 9.2.

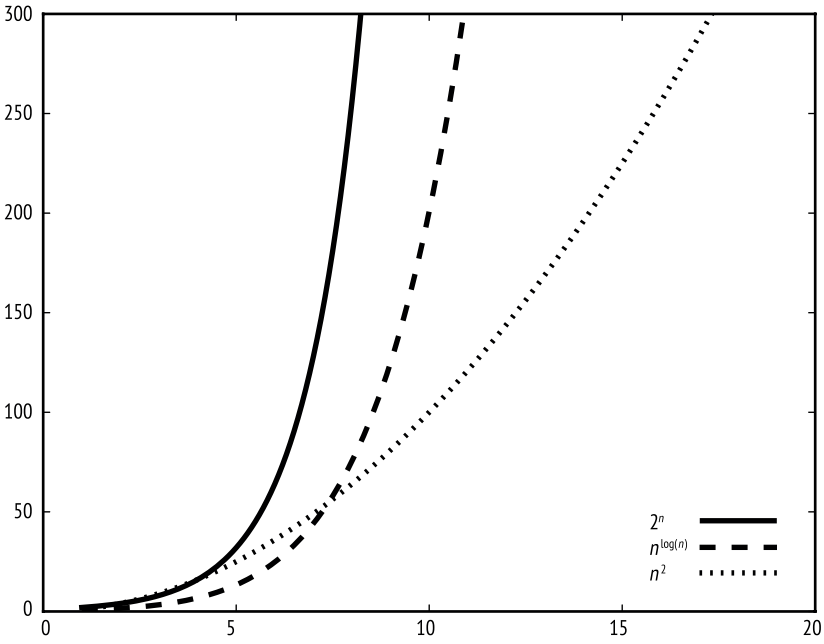

Рис. 9.2. Функции, растущие как 2^n, $n^{log(n)}$ и n^2, в порядке от самой быстрой к самой медленной

Примечание *Экспоненциальная сложность $O(2^n)$ – не худшее, с чем можно столкнуться. Есть сложности, растущие еще быстрее и, следовательно, характеризующие еще более медленные алгоритмы, например $O(n^n)$ или экспоненциальный факториал $O(n^{f(n-1)})$, где функция f ре-*

курсивно определена как $f(x) = x^{f(x-1)}$ *для любого x. Впрочем, на практике вам никогда не встретится алгоритм с такой абсурдной сложностью.*

Алгоритм сложности $O(n^2)$ или $O(n^3)$ еще может считаться эффективным, но $O(n^{99999999999})$, очевидно, нет. Иными словами, полиномиальное время приемлемо, если показатель степени не слишком велик. По счастью, у всех известных полиномиальных алгоритмов решения реальных задач показатели степени невелики. Например, алгоритм умножения двух n-битовых целых чисел имеет сложность $O(n^{1.465})$, а алгоритм умножения двух матриц $n \times n$ – сложность $O(n^{2.373})$. В 2002 году был открыт прорывной полиномиальный алгоритм нахождения простых чисел, сложность которого первоначально составляла $O(n^{12})$, но затем была улучшена до $O(n^6)$. Таким образом, полиномиальное время – возможно, не идеальное определение практически осуществимого, но это лучшее из того, что мы имеем.

Развивая эту мысль, мы считаем практически неразрешимой, или *трудной*, задачу, которую нельзя решить за полиномиальное время. Например, в случае прямолинейного поиска ключа невозможно справиться со сложностью $O(2^n)$, если только шифр не удастся каким-то образом взломать.

Мы точно знаем, что сложность $O(2^n)$ поиска ключа полным перебором непреодолима (при условии что шифр безопасен), но не всегда знаем самый быстрый способ решения задачи. Значительная часть исследований по теории сложности посвящена установлению *границ* времени выполнения алгоритмов решения данной задачи. Чтобы упростить себе работу, теоретики различают несколько групп, или *классов*, задач по усилиям, которые необходимо предпринять для их решения.

Классы сложности

В математике *классом* называется группа объектов с похожим свойством. Например, все вычислительные задачи, разрешимые за время $O(n^2)$, – множество, обозначаемое в теории сложности **TIME**(n^2), – составляют один класс. Аналогично **TIME**(n^3) – класс задач, разрешимых за время $O(n^3)$, **TIME**(2^n) – класс задач, разрешимых за время $O(2^n)$, и так далее. По той же причине, по которой суперкомпьютер может вычислить все, что способен вычислить ноутбук, любая задача, разрешимая за время $O(n^2)$, разрешима также за время $O(n^3)$. Поэтому любая задача класса **TIME**(n^2) принадлежит также классу **TIME**(n^3), и оба они являются подмножествами **TIME**(n^4) и т. д. Объединение всех классов **TIME**(n^k), где k – константа, обозначается **P** (полиномиальное время).

Если вы когда-нибудь программировали, то знаете, что кажущиеся быстрыми алгоритмы могут обрушить систему, потребив всю доступную память. Поэтому при выборе алгоритма нужно учитывать не только *временную*, но и *пространственную сложность* – сколько памяти необходимо алгоритму. Это тем более важно, что одно обращение

к памяти обычно занимает на несколько порядков больше времени, чем выполнение простой арифметической операции.

Формально потребление памяти алгоритмом можно определить как функцию от размера входных данных n точно так же, как мы определили временную сложность. Класс задач, разрешимых при использовании $f(n)$ бит памяти, обозначается **SPACE**($f(n)$). Например, **SPACE**(n^3) – класс задач, разрешимых при использовании порядка n^3 бит памяти. По аналогии с обозначением **P** для объединения всех классов **TIME**(n^k), мы обозначаем **PSPACE** объединение классов **SPACE**(n^k).

Очевидно, что чем меньше потребление памяти, тем лучше, но полиномиальный объем памяти еще не значит, что алгоритм практически осуществим. Почему? Возьмем, к примеру, поиск полным перебором: да, он потребляет пренебрежимо мало памяти, но при этом чудовищно медленный. Вообще, алгоритм может работать вечно, даже если для работы нужно всего несколько байтов памяти.

Любая задача, разрешимая за время $f(n)$, нуждается самое большее в памяти объемом $f(n)$, поэтому класс **TIME**($f(n)$) входит в **SPACE**($f(n)$). За время $f(n)$ мы сможем записать не более $f(n)$ бит, поскольку предполагается, что запись (или чтение) одного бита занимает одну единицу времени; поэтому любая задача, принадлежащая классу **TIME**($f(n)$), не может потребить память более $f(n)$ бит. Следовательно, **P** является подмножеством **PSPACE**.

Недетерминированное полиномиальное время

NP – второй по важности класс сложности после класса **P** всех алгоритмов с полиномиальным временем. Нет, **NP** означает не «неполиномиальное время», а «*недетерминированное* полиномиальное время». Что это такое?

NP – класс задач, для которых решение может быть проверено за полиномиальное время, т. е. эффективно, пусть даже найти решение может быть трудно. Говоря *проверено*, я имею в виду, что если дано потенциальное решение, то существует алгоритм с полиномиальным временем, который позволяет проверить, действительно ли это решение. Например, задача восстановления секретного ключа при наличии известного открытого текста принадлежит классу **NP**, потому что если даны P, $C = \mathbf{E}(K, P)$ и потенциальный ключ K_0, то можно проверить, является ли K_0 правильным ключом, убедившись, что $\mathbf{E}(K_0, P)$ равно C. Процесс нахождения потенциального ключа (решения) невозможно выполнить за полиномиальное время, но проверить, правилен ли ключ, можно с помощью полиномиального алгоритма.

А теперь контрпример: как насчет атак с известным шифртекстом? На этот раз у нас есть только значения $\mathbf{E}(K, P)$ для случайных неизвестных открытых текстов P. Если P неизвестны, то нет никакого способа проверить, является ли потенциальный ключ K_0 правильным. Иными словами, задача восстановления ключа в атаках с известным шифртекстом не принадлежит классу **NP** (и уж тем более **P**).

Еще один пример задачи, не принадлежащей классу **NP**, – проверка *отсутствия* решения у данной задачи. Проверка правильности решения сводится к выполнению некоторого алгоритма, получающего потенциальное решение на входе, и последующей проверке возвращенного значения. Но чтобы убедиться в *отсутствии* решения, возможно, придется перебрать все возможные входы. А если их количество экспоненциально велико, то эффективно доказать отсутствие решения не получится. Отсутствие решения трудно продемонстрировать для самых трудных задач из класса **NP** – так называемых **NP**-полных задач, к обсуждению которых мы и переходим.

NP-полные задачи

Самые трудные задачи в классе **NP** называются **NP**-полными; мы не знаем, как решить их за полиномиальное время. И как установили ученые в 1970-е годы, когда разрабатывалась теория **NP**-сложности, все самые трудные задачи из класса **NP** одинаково трудны. Для доказательства этого факта было показано, что любое эффективное решение любой **NP**-полной задачи можно преобразовать в эффективное решение любой другой **NP**-полной задачи. Иначе говоря, если мы умеем эффективно решать какую-нибудь **NP**-полную задачу, то можем решить любую из них, а значит, и все задачи класса **NP**. Как такое может быть?

NP-полные задачи могут рядиться в разные личины, но все они с математической точки зрения фундаментально похожи. На самом деле любую **NP**-полную задачу можно свести к любой другой **NP**-полной задаче, так что решение первой будет опираться на решение второй.

Вот несколько примеров **NP**-полных задач.

Задача коммивояжера. Дано множество точек на карте (города, адреса или иные географические местоположения) и попарные расстояния между точками. Требуется найти маршрут, проходящий через каждую точку и такой, что его суммарная длина меньше заданной величины *x*.

Задача о клике. Дано число *x* и граф (множество вершин, соединенных ребрами, как на рис. 9.3). Требуется определить, существует ли в нем *x* или менее вершин, попарно соединенных между собой.

Задача о рюкзаке. Даны два числа, *x* и *y*, и множество предметов известной ценности и веса. Спрашивается, можно ли выбрать несколько предметов, суммарная ценность которых будет не менее *x*, а суммарный вес не более *y*?

Такие **NP**-полные задачи встречаются везде, от задач планирования (даны работы с известным приоритетом и длительностью и один или несколько процессоров, требуется распределить работы по процессорам с соблюдением приоритетов, так чтобы минимизировать общее время выполнения) до задач удовлетворения ограничений

(найти значения, удовлетворяющие множеству математических ограничений, например логических уравнений). **NP**-полноту можно доказать даже для задач выигрыша в некоторых видеоиграх (допустим, *Tetris*, *Super Mario Bros.*, *Pokémon* и *Candy Crush Saga*). Например, в статье «Classic Nintendo Games Are (Computationally) Hard» (*https://arxiv.org/abs/1203.1895*) рассматривается «задача принятия решения о достижимости», цель которой – найти вероятность достижения конечной точки из заданной начальной точки.

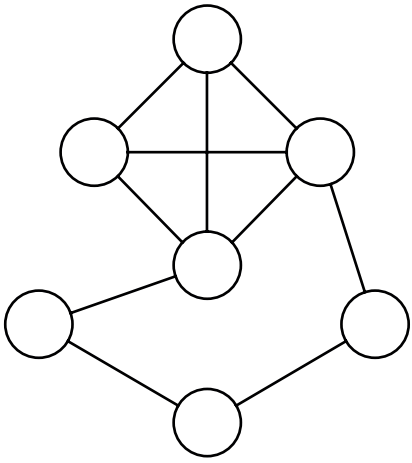

*Рис. 9.3. Граф, содержащий клику из четырех вершин. Общая задача нахождения клики (множества вершин, в котором каждая вершина соединена ребром с каждой) заданного размера является **NP**-полной*

Некоторые из этих задач в видеоиграх даже труднее **NP**-полных, они называются **NP**-*трудными*. Говорят, что задача **NP**-трудная, если она по меньшей мере так же трудна, как **NP**-полные задачи. Формально задача называется **NP**-трудной, если можно доказать, что, решив ее, мы решим также **NP**-полные задачи.

Я должен упомянуть важную тонкость. Не все *экземпляры* **NP**-полной задачи одинаково трудно решить. Некоторые экземпляры можно решить эффективно, потому что они малы по размеру или имеют специальную структуру. Взять, к примеру, граф на рис. 9.3. Достаточно посмотреть на него несколько секунд, чтобы найти клику – четыре верхние вершины, попарно соединенные ребрами; хотя вышеупомянутая задача о клике **NP**-трудная, в данном случае ничего трудного нет. Так что **NP**-полнота не означает, что все экземпляры данной задачи трудны, а лишь то, что с ростом размера задачи многие оказываются таковыми.

Задача о равенстве P и NP

Если бы можно было решить самые трудные задачи класса **NP** за полиномиальное время, то за полиномиальное время можно было бы решить *все* **NP**-задачи, поэтому класс **NP** совпадал бы с классом **P**. Это звучит абсурдно: разве не очевидно, что существуют задачи, решение которых легко проверить, но трудно найти? Например, разве не

очевидно, что экспоненциальный полный перебор – самый быстрый способ восстановить ключ симметричного шифра и что поэтому данная задача не может принадлежать классу **P**? Но, как ни странно, до сих пор никто не смог доказать, что класс **P** отличается от **NP**, несмотря на объявленный приз в миллион долларов.

Математический институт Клэя посулил эту награду любому, кто сможет доказать, что **P** ≠ **NP** или что **P** = **NP**. Известный специалист по теории сложности Скотт Ааронсон назвал эту задачу о равенстве **P** и **NP** «одним из самых глубоких вопросов, когда-либо заданных человеком». Вдумайтесь: если бы **P** был равен **NP**, то любое легко проверяемое решение было бы легко найти. Вся практическая криптография оказалась бы небезопасной, потому что мы смогли бы эффективно восстанавливать ключи симметричных шифров и обращать функции хеширования.

Но не впадайте в панику: большинство специалистов по теории сложности полагают, что **P** не равен **NP**, а является строгим подмножеством **NP**, как показано на рис. 9.4, где **NP**-полные задачи представляют собой еще одно подмножество **NP**, не пересекающееся с **P**. Иначе говоря, задачи, которые выглядят сложными, являются таковыми на самом деле. Просто это трудно доказать математически. Для доказательства того, что **P** = **NP**, нужно было бы только предъявить полиномиальный алгоритм для любой **NP**-полной задачи, а вот доказать, что такого алгоритма не существует, принципиально труднее. Но это не помешало математикам-любителям публиковать простые доказательства, которые, несмотря на очевидную некорректность, часто бывает забавно читать; см., например, страницу «P-versus-NP» (*https://www.win.tue.nl/~gwoegi/P-versus-NP.htm*).

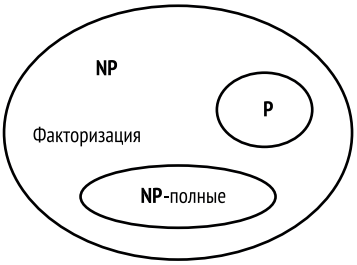

*Рис. 9.4. Классы **NP**, **P** и множество **NP**-полных задач*

Но если мы почти уверены, что трудные задачи существуют, почему бы не использовать их для построения стойкой, доказуемо безопасной криптографии? Представьте, что имеется доказательство того, что взлом некоторого шифра – **NP**-полная задача, тогда этот шифр можно считать невзламываемым при условии, что **P** не равен **NP**. Но действительность разочаровывает: доказано, что **NP**-полные задачи трудно использовать для криптографических целей, потому что та самая структура, которая делает их трудными в общем случае, открывает возможность простого решения в частных случаях, а они иногда встречаются в криптографии. Поэтому криптографы часто полагаются на задачи, которые, *вероятно*, *не* являются **NP**-трудными.

Задача факторизации

Задача факторизации заключается в нахождении простых чисел p и q по заданному большому числу $N = p \times q$. Широко применяемые алгоритмы RSA основаны на том факте, что факторизация числа – трудная задача. На самом деле именно трудность задачи факторизации и делает схемы шифрования и цифровой подписи RSA безопасными. Но прежде чем показать, как в RSA используется задача факторизации (этому посвящена глава 10), я хочу убедить вас, что эта задача действительно трудная, хотя, вероятно, и не **NP**-полная.

Сначала немного математики для младших школьников. *Простым* называется число, которое делится только на себя и на 1. Например, числа 3, 7, 11 простые, а числа $4 = 2 \times 2$, $6 = 2 \times 3$ и $12 = 2 \times 2 \times 3$ непростые. Основная теорема арифметики утверждает, что любое целое число можно единственным образом представить в виде произведения простых чисел; это представление называется разложением числа на множители, или *факторизацией*. Например, факторизация 123 456 имеет вид $2^6 \times 3 \times 643$, факторизация 1 234 567 – 127×9721 и т. д. Для любого целого числа факторизация единственна, т. е. его можно записать в виде произведения простых чисел только одним способом. Но откуда нам знать, что данная факторизация состоит только из простых чисел? Иначе говоря, откуда мы знаем, что данное число простое? Ответ дают алгоритмы проверки на простоту, имеющие полиномиальное время работы. Однако переход от числа к его простым множителям – совсем другое дело.

Факторизация больших чисел на практике

Итак, как перейти от числа N к его факторизации, т. е. к разложению на простые множители? Самый простой способ – пробовать все числа, меньшие N, пока не найдется число x, на которое N делится нацело. Затем попробовать следующее число $x + 1$ и т. д. В итоге получим список множителей N. Какова временная сложность такого алгоритма? Для начала вспомним, что сложность выражается в виде функции от *длины* входных данных. Длина числа N в битах равна $n = \log_2 N$. Согласно определению логарифма, отсюда следует, что $N = 2^n$. Поскольку все числа, меньшие $N/2$, могут считаться возможными множителями N, нам придется перепробовать приблизительно $N/2 = 2^n/2$ значений. Следовательно, сложность нашего наивного алгоритма факторизации составляет $O(2^n)$, поскольку коэффициент 1/2 в нотации $O()$ можно игнорировать.

Разумеется, есть много чисел, которые легко факторизовать, определив сначала небольшие простые множители (2, 3, 5 и т. д.), а затем итеративно факторизуя остальные непростые множители. Но в данном случае нас интересуют числа вида $N = p \times q$, где p и q велики.

Поступим немного умнее. Нам не нужно проверять все числа, меньшие $N/2$, а только те из них, которые являются простыми. И при этом можно проверять только числа, меньшие квадратного корня из N.

Действительно, если N – непростое число, то у него должен быть хотя бы один множитель, меньший $\sqrt{N}$, поскольку если бы оба множителя p и q были больше $\sqrt{N}$, то их произведение было бы больше $\sqrt{N} \times \sqrt{N} = N$. Так, если $N = 100$, то оба множителя p и q не могут быть больше 10, потому что иначе их произведение было бы больше 100. Либо p, либо q должно быть меньше $\sqrt{N}$.

А какова сложность проверки только простых чисел, меньших $\sqrt{N}$? *Теорема о распределении простых чисел* утверждает, что существует приблизительно $N/\log N$ простых чисел, меньших N. Следовательно, существует приблизительно $\sqrt{N}/\log\sqrt{N}$ простых чисел, меньших $\sqrt{N}$. Поскольку $\sqrt{N} = 2^{n/2}$, а $1/\log\sqrt{N} = 1/(n/2) = 2n$, получается, что существует приблизительно $2^{n/2}/n$ возможных простых множителей и, значит, сложность равна $O(2^{n/2}/n)$. Это быстрее, чем проверять все простые числа, но все равно мучительно медленно – порядка 2^{120} операций для 256-битового числа. Совершенно неподъемное вычисление.

Самый быстрый алгоритм факторизации – *общий метод решета числового поля* (general number field sieve – GNFS), который я не буду здесь описывать, потому что для его понимания требуется владение нетривиальным математическим аппаратом. Грубая оценка сложности GNFS – $\exp(1.91 \times n^{1/3}(\log n)^{2/3})$, где $\exp(\ldots)$ – другое обозначение экспоненциальной функции e^x, а e – постоянная Эйлера, равная приблизительно 2.718. Но получить точную оценку сложности GNFS для заданного размера числа трудно. Поэтому приходится опираться на эвристические оценки, показывающие, как растет безопасность при увеличении n. Например:

- для факторизации **1024-битового** числа, имеющего два простых множителя длиной примерно по 500 бит, требуется порядка 2^{70} элементарных операций;
- для факторизации **2048-битового** числа, имеющего два простых множителя длиной примерно по 1000 бит, требуется порядка 2^{90} элементарных операций, т. е. приблизительно в миллион раз больше, чем для 1024-битового числа.

Согласно имеющимся оценкам, для достижения 128-битовой безопасности число должно содержать по меньшей мере 4096 бит. Заметим, что к этим оценкам следует относиться с долей скептицизма, и не все исследователи с ними согласны. Взгляните на экспериментальные результаты, демонстрирующие фактическую стоимость факторизации:

- в 2005 году, после примерно 18 месяцев вычислений на кластере из 80 процессоров (эквивалент 75 лет вычислений на одном процессоре) удалось факторизовать **663-битовое** (200 десятичных цифр) число;
- в 2009 году после двух лет вычислений с использованием нескольких сотен процессоров, что эквивалентно примерно 2000 лет вычислений на одном процессоре, другая группа исследователей факторизовала **768-битовое** (232 десятичные цифры) число.

Как видим, числа, которые удалось факторизовать, короче применяемых в реальных приложениях, где длина составляет не менее 1024, а зачастую более 2048 бит. На момент написания этой книги не было сообщений о факторизации 1024-битового числа, но многие думают, что хорошо финансируемые организации типа АНБ способны это сделать.

Короче говоря, 1024-битовый алгоритм RSA следует считать небезопасным, а для достижения более высокого уровня безопасности RSA нужно использовать с числом, содержащим не менее 2048 бит, а лучше 4096 бит.

Является ли задача факторизации NP-полной?

Мы не знаем эффективного способа факторизации больших чисел, поэтому можно предположить, что задача факторизации не принадлежит классу **P**. Однако она, безусловно, принадлежит классу **NP**, потому что для проверки правильности предложенной факторизации достаточно с помощью вышеупомянутого алгоритма проверки на простоту убедиться, что все множители простые и что их произведение равно заданному числу. Например, чтобы проверить, что 3×5 – факторизация 15, нужно убедиться, что 3 и 5 – простые числа и что произведение 3 и 5 равно 15.

Итак, у нас есть задача, принадлежащая классу **NP**, которая кажется трудной, но так ли она трудна, как самые трудные **NP**-задачи? Иными словами, является ли задача факторизации **NP**-полной? Предупреждение: вероятно, нет.

Математически не доказано, что задача факторизации не является **NP**-полной, но есть косвенные свидетельства. Во-первых, все известные **NP**-полные задачи могут иметь одно решение, но бывает также, что решений много или нет ни одного. А задача факторизации всегда имеет ровно одно решение. Кроме того, у задачи факторизации есть математическая структура, благодаря которой алгоритм GNFS работает намного быстрее наивного. А у **NP**-полных задач такой структуры нет. Выполнить факторизацию было бы просто при наличии *квантового компьютера*, вычислительной модели, в которой для выполнения алгоритмов разных видов используются квантово-механические явления и которая позволила бы эффективно разлагать на множители большие числа (не потому, что алгоритм работал бы быстрее, а потому, что имеется квантовый алгоритм, специально заточенный на факторизацию больших чисел). Впрочем, пока квантовых компьютеров не существует, и, быть может, они никогда и не появятся. Как бы то ни было, квантовый компьютер был бы бесполезен для решения **NP**-полных задач, потому что он работал бы не быстрее классического (см. главу 14).

Итак, задача факторизации теоретически, возможно, немного проще, чем **NP**-полные задачи, но с точки зрения криптографии она достаточно трудна и даже надежнее **NP**-полных задач. Действительно, построить криптосистемы на базе задачи факторизации легче, чем на базе **NP**-полных задач, потому что заранее трудно сказать, насколько

сложно будет взломать криптосистему на базе какой-то **NP**-полной задачи, т. е. сколько битов безопасности она обеспечивает.

Задача факторизации – лишь одна из нескольких задач, играющих в криптографии роль *предположения о трудности*, т. е. предположения о том, что некоторая задача вычислительно трудна. Это предположение используется при доказательстве того, что взломать криптосистему так же трудно, как решить соответствующую задачу. Еще одна подобная задача – *задача о дискретном логарифме* (discrete logarithm problem – DLP); на самом деле это семейство задач, которые мы обсудим ниже.

Задача о дискретном логарифме

В официальной истории криптографии DLP предшествует задаче факторизации. Алгоритм RSA появился в 1977 году, а второй прорыв в криптографии, протокол совместной выработки ключа Диффи–Хеллмана, датируется годом раньше, и его безопасность основана на трудности DLP. Как и задача факторизации, DLP имеет дело с большими числами, но он не столь очевиден – чтобы ухватить его суть, понадобится несколько минут, а не секунд, да и математики он требует больше. Итак, давайте познакомимся с математическим понятием *группы* в контексте дискретных логарифмов.

Что такое группа?

В математике *группой* называется множество элементов (чаще всего чисел), для которого определены некоторые правила. Примером группы является множество ненулевых целых чисел (от 1 до $p - 1$) по модулю некоторого простого числа p; эта группа обозначается $\mathbf{Z}_p^*$. Для $p = 5$ получается группа $\mathbf{Z}_5^* = \{1,2,3,4\}$. В группе $\mathbf{Z}_5^*$ операции выполняются по модулю 5, поэтому 3×4 равно не 12, а 2, т. к. $12 \bmod 5 = 2$. Тем не менее мы используем тот же знак (×), что и для обычного умножения целых чисел. И точно так же для обозначения умножения элемента группы на себя по модулю p (частая операция в криптографии) употребляется обычная нотация возведения в степень. Например, в группе $\mathbf{Z}_5^*$ $2^3 = 2 \times 2 \times 2 = 3$, а не 8, потому что $8 \bmod 5 = 3$.

Чтобы быть группой, множество должно обладать следующими характеристиками, называемыми *аксиомами группы*:

- **замкнутость.** Для любых двух элементов группы x и y произведение $x \times y$ также принадлежит группе. В $\mathbf{Z}_5^*$ $2 \times 3 = 1$ (потому что $6 = 1 \bmod 5$), $2 \times 4 = 3$ и т. д.;
- **ассоциативность.** Для любых элементов группы x, y, z имеет место равенство $(x \times y) \times z = x \times (y \times z)$. В $\mathbf{Z}_5^*$ $(2 \times 3) \times 4 = 1 \times 4 = 2 \times (3 \times 4) = 2 \times 2 = 4$;
- **существование единичного элемента.** Существует такой элемент e, что $e \times x = x \times e = x$. В любой группе $\mathbf{Z}_p^*$ единичным элементом является 1;

- **существование обратного элемента.** Для любого элемента группы x существует элемент y такой, что $x \times y = y \times x = e$. В $\mathbf{Z}_5^*$ элементом, обратным к 2, является 3, а элементом, обратным к 3, является 2, тогда как 4 является обратным самому себе, т. к. $4 \times 4 = 16 = 1 \bmod 5$.

Группа называется *коммутативной*, если $x \times y = y \times x$ для любых элементов x и y. Этим свойством обладает любая мультипликативная группа целых чисел $\mathbf{Z}_p^*$. В частности, $\mathbf{Z}_5^*$ коммутативна: $3 \times 4 = 4 \times 3$, $2 \times 3 = 3 \times 2$ и т. д.

Группа называется *циклической*, если существует хотя бы один элемент g такой, что множество его степеней (g^1, g^2, g^3 и т. д.) по модулю p содержит все элементы группы. Сам элемент g в этом случае называется *порождающим элементом*, или *генератором* группы. Группа $\mathbf{Z}_5^*$ циклическая и имеет два порождающих элемента, 2 и 3, потому что $2^1 = 2$, $2^2 = 4$, $2^3 = 3$, $2^4 = 1$ и $3^1 = 3$, $3^2 = 4$, $3^3 = 2$, $3^4 = 1$.

Я использовал в качестве группового оператора умножение, но существуют группы и с другими операторами. Например, самая очевидная группа – это множество всех целых чисел, положительных и отрицательных, относительно операции сложения. Проверим, что для нее выполняются аксиомы группы. Очевидно, что число $x + y$ целое, если x и y целые (замкнутость); $(x + y) + z = x + (y + z)$ для любых x, y, z (ассоциативность); единичным элементом является 0, а обратным к любому элементу группы x является $-x$, потому что $x + (-x) = 0$ для любого целого x. Но есть большая разница: группа целых чисел бесконечная, тогда как в криптографии нас интересуют только *конечные группы*, имеющие конечное число элементов. Обычно используются группы $\mathbf{Z}_p^*$, где p насчитывает *тысячи* битов (такие группы содержат порядка 2^p чисел).

Трудная задача

В задаче о дискретном логарифме задано основание g и $x \in \mathbf{Z}_p^*$, где p – простое число, а требуется найти такое y, для которого $g^y = x$ в $\mathbf{Z}_p^*$. Слово *дискретный* употребляется, потому что мы имеем дело с целыми, а не с вещественными (непрерывными) числами, а *логарифм* – потому что мы ищем логарифм x по основанию g. (Например, логарифм 256 по основанию 2 равен 8, поскольку $2^8 = 256$.)

Меня часто спрашивают: что безопаснее – факторизация или дискретное логарифмирование? Иными словами, какая задача труднее. Я отвечаю, что их трудность приблизительно одинакова. На самом деле алгоритмы решения DLP в чем-то похожи на алгоритмы факторизации целых чисел, и уровень безопасности для n-битовых чисел, трудных для факторизации, примерно такой же, как для дискретных логарифмов в n-битовой группе. И по той же причине, что задача факторизации, DLP не является **NP**-полной. (Заметим, что в некоторых группах решить DLP проще, но я сейчас говорю только о группах по простому модулю).

Какие возможны проблемы

Прошло уже больше 40 лет, а мы до сих пор не знаем, как эффективно раскладывать большие числа на множители или находить дискретные логарифмы. Любители могут возразить, что кто-нибудь когда-нибудь решит задачу факторизации, – и у нас нет доказательства, что это невозможно, – но ведь и доказывать, что **P** ≠ **NP**, мы тоже не умеем. Можно было бы порассуждать на тему того, что **P** может быть равно **NP**, однако, по мнению специалистов, это крайне маловероятно. Так что нет причин для беспокойства. И действительно, вся современная криптография с открытым ключом опирается либо на факторизацию (RSA), либо на DLP (протокол Диффи–Хеллмана, схема Эль-Гамаля, эллиптическая криптография). Но хотя математика нас не подведет, могут вмешаться проблемы реального мира и человеческие ошибки.

Когда разложить на множители легко

Факторизовать большие числа не всегда трудно. Возьмем, к примеру, следующее 1024-битовое число N:

179769313486231590772930519078902473361797697894230657273430081157739
343819933842986982557174198257278917258638193709265819186026626180659
730665062710995556578639447715608415186895652841691982921107202317165
369124890481512388558039053427125099290315449262324709315263256083132
5404614070528728327909153880145 92

Для 1024-битовых чисел $N = pq$, используемых при шифровании и схемах цифровой подписи на основе RSA, мы ожидаем, что лучшие алгоритмы факторизации должны будут выполнить порядка 2^{70} операций. Но это конкретное число можно разложить на множители за секунды, воспользовавшись написанной на Python математической библиотекой SageMath. Функция `factor()` на моем компьютере MacBook 2015 года выпуска нашла следующее разложение меньше, чем за пять секунд:

$$2^{800} \times 641 \times 6700417 \times 167773885276849215533569 \times 37414057161322375957408148834323969.$$

Да, я схитрил. Это число не имеет вид $N = pq$, у него не два простых сомножителя, а пять, в т. ч. очень маленькие, благодаря чему его легко разложить на множители. Сначала мы находим часть $2^{800} \times 641 \times 6700417$, перепробовав небольшие простые числа из заранее вычисленного списка, после чего остается 192-битовое число, которое факторизовать гораздо проще, чем 1024-битовое число с двумя большими множителями.

Но факторизация может оказаться простой не только тогда, когда N имеет малые простые множители, но и тогда, когда N или его множи-

тели p и q имеют определенный вид – например, когда $N = pq$, причем p и q близки к некоторой степени двойки 2^b, или когда $N = pq$ и некоторые биты p или q известны, или когда N имеет вид $N = p^r q^s$, где $r > \log p$. Однако подробное обсуждение причин таких слабостей выходит за рамки этой книги.

Вывод: алгоритмы шифрования и цифровой подписи на основе RSA (рассматриваются в главе 10) должны работать со значением $N = pq$, где p и q тщательно подобраны, чтобы факторизация N не была легкой, иначе возможно катастрофическое нарушение безопасности.

Небольшие трудные задачи трудными не являются

Вычислительно трудные задачи становятся легкими, если они достаточно малы – даже алгоритмы с экспоненциальным временем становятся практически пригодными, если уменьшить размер задачи. Симметричный шифр может быть безопасным в том смысле, что не существует атаки полным перебором менее 2^n вариантов, но если длина ключа $n = 32$, то для взлома шифра хватит нескольких минут. Это кажется очевидным, и вы, наверное, думаете, что никто в здравом уме не станет использовать короткие ключи, но в реальности есть немало причин, по которым такое возможно. Расскажу две правдивые истории.

Представьте себе разработчика, который ничего не знает о криптографии, но располагает каким-то API шифрования с помощью RSA и получил задание зашифровать сообщение с безопасностью 128 бит. Какой размер ключа RSA он выберет? Я встречал ситуации, когда выбирался 128-битовый RSA, т. е. основанный на 128-битовом числе $N = pq$. Да, факторизация числа N длиной в тысячи бит практически неосуществима, но разложить на множители 128-битовое число легко. Команды библиотеки SageMath, показанные в листинге 9.2, выполняются мгновенно.

Листинг 9.2. Генерирование модуля RSA путем выбора двух случайных простых чисел с последующей мгновенной факторизацией

```
sage: p = random_prime(2**64)
sage: q = random_prime(2**64)
sage: factor(p*q)
6822485253121677229 * 17596998848870549923
```

Листинг 9.2 показывает, что 128-битовое число, равное произведению двух случайно выбранных 64-битовых простых чисел, легко факторизовать даже на типичном ноутбуке. Но если бы я выбрал 1024-битовые простые числа, `p = random_prime(2**1024)`, то команда `factor(p*q)` никогда не завершилась бы, во всяком случае я бы до этого не дожил.

Откровенно говоря, имеющиеся инструменты не помогают предотвратить наивное использование опасно коротких параметров. На-

пример, пакет OpenSSL позволяет генерировать ключи RSA длиной 31 бит, не выдавая предупреждений; очевидно, что такие короткие ключи, показанные в листинге 9.3, абсолютно небезопасны.

Листинг 9.3. Генерирование небезопасного закрытого ключа RSA с помощью пакета OpenSSL

```
$ openssl genrsa 31
Generating RSA private key, 31 bit long modulus
.++++++++++++++++++++++++++++
.++++++++++++++++++++++++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBHHqFuUCAwEAAQIEP6zEJQIDANATAgMAjCcCAwCSBwICTGsCAhpp
-----END RSA PRIVATE KEY-----
```

Просматривая криптографический код, следует обращать внимание не только на типы используемых алгоритмов, но и на их параметры и на длины секретных значений. Однако, как показывает следующая история, то, что считается безопасным сегодня, может оказаться небезопасным завтра.

В 2015 году исследователи обнаружили, что многие HTTPS- и почтовые серверы все еще поддерживают устаревшую небезопасную версию протокола выработки ключа Диффи–Хеллмана. Точнее, используемая реализация TLS поддерживала протокол Диффи–Хеллмана в группе $\mathbf{Z}_p^*$ с простым числом p длиной всего 512 бит, для которой задача дискретного логарифмирования уже не считалась практически неразрешимой.

Мало того что серверы поддерживали слабый алгоритм, так еще противник мог заставить благонамеренного клиента использовать этот алгоритм, внедрив вредоносный трафик в клиентский сеанс. При этом большую часть атаки можно было организовать один раз и повторять для воздействия на несколько клиентов – просто рай для противника. После примерно недели вычислений для атаки на конкретную группу $\mathbf{Z}_p^*$ взлом сеансов различных пользователей занимал всего 70 секунд.

Безопасный протокол не стоит ни гроша, если его сводит на нет ставший нестойким алгоритм, а надежный алгоритм бесполезен, если используется со слабыми параметрами. В криптографии всегда нужно читать написанное мелким шрифтом.

Дополнительные сведения об этой истории приведены в статье «Imperfect Forward Secrecy: How Diffie–Hellman Fails in Practice» (*https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf*).

Для дополнительного чтения

Призываю вас глубже познакомиться с фундаментальными аспектами вычислений в контексте теорий вычислимости (какие функ-

ции можно вычислить?) и сложности (с какими затратами?), а также изучить их связи с криптографией. Я говорил в основном о классах **P** и **NP**, но существует много других классов и моментов, представляющих интерес для криптографов. Настоятельно рекомендую книгу Scott Aaronson «Quantum Computing Since Democritus» (Cambridge University Press, 2013). Большая ее часть посвящена квантовым вычислениям, но в первых главах блестяще изложены основы теории сложности и криптографии.

В научной литературе по криптографии можно встретить и другие трудные вычислительные задачи. Я упомяну о них в следующих главах, но уже сейчас приведу несколько примеров, иллюстрирующих разнообразие задач, которые используются криптографами в своих целях.

- Задача Диффи–Хеллмана (зная g^x и g^y, найти g^{xy}) – вариант задачи о дискретном логарифме, который широко используется в протоколах совместной выработки ключей.
- Задачи теории решеток, например задача о нахождении кратчайшего вектора (shortest vector problem – SVP) и задача обучения с ошибками (learning with errors – LWE), – единственные примеры **NP**-трудных задач, успешно используемых в криптографии.
- Задачи кодирования опираются на трудность декодирования кодов, исправляющих ошибки, при недостатке информации. Они изучаются с конца 1970-х годов.
- Многомерные задачи возникают при решении системы нелинейных уравнений. Потенциально они **NP**-трудные, но на их основе не удалось построить надежных криптосистем, потому что трудные версии велики и работают медленно, а практически пригодные оказались небезопасными.

В главе 10 мы продолжим разговор о трудных задачах, особенно о факторизации и ее главном варианте, задаче RSA.

10

RSA

Криптосистема Ривеста–Шамира–Адлемана (Rivest–Shamir–Adleman – RSA), появившаяся в 1977 году, совершила революцию в криптографии, став первой схемой шифрования с открытым ключом. Если в классических схемах шифрования с симметричным ключом один и тот же секретный ключ использовался как для шифрования, так и для дешифрирования сообщений, то в схеме шифрования с открытым ключом (ее также называют *асимметричным* шифрованием) используется два ключа: открытый, которым может воспользоваться любой желающий, чтобы зашифровать сообщение для вас, и закрытый, который необходим для дешифрирования сообщений, зашифрованных открытым ключом. Благодаря этому волшебству RSA стала настоящим прорывом и 40 лет спустя по-прежнему является образцом шифрования с открытым ключом и рабочей лошадкой безопасности в интернете. (За год до появления RSA Диффи и Хеллман предложили идею криптографии с открытым ключом, но их схема оказалась неспособной к подписанию с открытым ключом.)

RSA – это прежде всего арифметический трюк. В основе ее работы лежит математический объект, называемый *перестановкой с потайным входом* (trapdoor permutation), – функция, которая преобразует число x в число y в том же диапазоне, так что вычислить y по x легко, зная открытый ключ, но вычислить x по y практически невозможно, если не знать закрытого ключа – *потайного входа*. (Можете считать, что x – открытый текст, а y – шифртекст.)

Кроме шифрования, RSA используется для создания цифровых подписей, когда только владелец закрытого ключа может подписать сообщение, а наличие открытого ключа позволяет любому желающему проверить достоверность подписи.

В этой главе я объясню, как работает перестановка с потайным входом в RSA, разберу, как безопасность RSA связана с задачей факторизации (см. главу 9), и расскажу, почему одной лишь перестановки с потайным входом недостаточно для построения *безопасных* шифрования и цифровой подписи. Я также рассмотрю способы реализации RSA и продемонстрирую атаки на нее.

Начнем с математических понятий, лежащих в основе RSA.

Математические основания RSA

Алгоритм RSA видит сообщение как большое число, а само шифрование заключается, по существу, в умножении больших чисел. Поэтому, чтобы понять, как работает система RSA, нужно знать, какого рода большими числами она манипулирует и как устроено умножение этих чисел.

RSA видит зашифровываемый открытый текст как положительное целое число от 1 до $n - 1$, где n – большое число, называемое *модулем*. При перемножении таких чисел получается третье число, удовлетворяющее тем же условиям. Мы говорим, что эти числа образуют группу, обозначаемую $\mathbf{Z}_n^*$ и называемую мультипликативной группой целых чисел по модулю n. (Математическое определение группы см. в разделе «Что такое группа?» главы 9.)

Например, рассмотрим группу $\mathbf{Z}_4^*$ целых чисел по модулю 4. Напомним, что группа должна содержать единичный элемент (т. е. 1) и что у каждого элемента группы должен быть обратный – число y, такое что $x \times y = 1$. Как найти множество, образующее $\mathbf{Z}_4^*$? Исходя из наших определений, мы знаем, что 0 не принадлежит группе $\mathbf{Z}_4^*$, потому что умножение любого числа на 0 не может дать 1, так что у 0 не было бы обратного элемента. С другой стороны, число 1 принадлежит $\mathbf{Z}_4^*$, поскольку $1 \times 1 = 1$, так что 1 обратна самой себе. Однако число 2 этой группе не принадлежит, т. к. невозможно получить 1, умножая любой элемент $\mathbf{Z}_4^*$ на 2 (причина в том, что 2 и 4 – не взаимно простые числа, поскольку у них есть общий множитель 2). Число 3 принадлежит группе элементов $\mathbf{Z}_4^*$, потому что оно обратно самому себе. Таким образом, элемент $\mathbf{Z}_4^* = \{1, 3\}$.

Теперь рассмотрим $\mathbf{Z}_5^*$, мультипликативную группу целых чисел по модулю 5. Из каких чисел она состоит? Число 5 простое, и все числа

1, 2, 3, 4 взаимно просты с 5, поэтому множество $\mathbf{Z}_5^*$ совпадает с {1, 2, 3, 4}. Проверим: $2 \times 3 \bmod 5 = 1$, поэтому 2 обратно к 3, а 3 обратно к 2; 4 обратно самому себе, потому что $4 \times 4 \bmod 5 = 1$; наконец, 1 – тоже самообратный элемент группы.

Для нахождения числа элементов группы $\mathbf{Z}_n^*$, когда n не является простым числом, используется *функция Эйлера* $\varphi(n)$. Она дает количество чисел, меньших n и взаимно простых с n, т. е. как раз количество элементов $\mathbf{Z}_n^*$. Если n разложить в произведение простых чисел $n = p_1 \times p_2 \times \ldots \times p_m$, то

$$\varphi(n) = (p_1 - 1) \times (p_2 - 1) \times \ldots \times (p_m - 1).$$

RSA имеет дело только с числами n, являющимися произведением двух больших простых чисел, $n = pq$. Соответствующая группа $\mathbf{Z}_n^*$ содержит $\varphi(n) = (p - 1)(q - 1)$ элементов. Раскрывая скобки, получаем эквивалентное определение $\varphi(n) = n - p - q + 1$, или $\varphi(n) = (n + 1) - (p + q)$, которое интуитивно более понятно выражает значение $\varphi(n)$ в терминах n.

Перестановка с потайным входом в RSA

Перестановка с потайным входом – базовый алгоритм, лежащий в основе шифрования и подписи в RSA-системах. Если задан модуль n и число e, называемое открытым показателем степени, то перестановка с потайным входом преобразует число $x \in \mathbf{Z}_n^*$ в число $y = x^e \bmod n$. При использовании перестановки с потайным входом для шифрования модуль n и показатель степени e составляют открытый ключ RSA.

Чтобы получить x по y, нам нужно еще одно число, d, такое что:

$$y^d \bmod n = (x')^d \bmod n = x^{ed} \bmod n = x.$$

Поскольку число d – потайной вход, позволяющий выполнять дешифрирование, оно является частью закрытого ключа RSA и, в отличие от открытого ключа, должно храниться в секрете. Число d называют также *секретным показателем степени*.

Очевидно, что d – не любое число, а такое, что e, умноженное на d, эквивалентно 1 и, следовательно, $x^{ed} \bmod n = x$ для любого x. Точнее, должно иметь место равенство $ed = 1 \bmod \varphi(n)$, тогда будет $x^{ed} = x^1 = x$, и мы сможем правильно дешифрировать сообщение. Заметим, что вычисление производится по модулю $\varphi(n)$, а не по модулю n, потому что показатели степени ведут себя как индексы элементов $\mathbf{Z}_n^*$, а не сами элементы. Поскольку $\mathbf{Z}_n^*$ содержит $\varphi(n)$ элементов, индекс должен быть меньше $\varphi(n)$.

Число $\varphi(n)$ критически важно для безопасности RSA. На самом деле нахождение $\varphi(n)$ для модуля RSA n эквивалентно взлому RSA, поскольку секретный показатель степени d легко вывести, зная $\varphi(n)$ и e, – достаточно просто вычислить элемент, обратный e. Поэтому p

и q должны храниться в секрете, т. к. знание того или другого позволяет определить $\varphi(n)$ по формуле $(p - 1)(q - 1) = \varphi(n)$.

Примечание *$\varphi(n)$ называют также порядком группы $\mathbf{Z}_n^*$; порядок – важная характеристика группы, являющаяся также неотъемлемой частью других систем с открытым ключом, например протокола Диффи–Хеллмана и эллиптической криптографии.*

Генерирование ключей и безопасность RSA

Под *генерированием ключей* понимается процедура создания пары ключей RSA, а именно открытого ключа (модуля n и открытого показателя степени e) и закрытого ключа (секретного показателя степени d). Числа p и q (такие, что $n = pq$) и порядок $\varphi(n)$ также следует хранить в секрете, поэтому они часто считаются частью закрытого ключа.

Чтобы сгенерировать пару ключей RSA, мы сначала выбираем два случайных простых числа p и q, а затем вычисляем по ним $\varphi(n)$ и элемент d, обратный e. Чтобы показать, как это работает, мы в листинге 10.1 воспользовались написанной на Python средой SageMath (*http://www.sagemath.org/*) с открытым исходным кодом, которая включает много математических пакетов.

Листинг 10.1. Генерирование параметров RSA с помощью SageMath

```
❶ sage: p = random_prime(2^32); p
  1103222539
❷ sage: q = random_prime(2^32); q
  17870599
❸ sage: n = p*q; n
  19715247602230861
❹ sage: phi = (p-1)*(q-1); phi
  19715246481137724
❺ sage: e = random_prime(phi); e
  13771927877214701
❻ sage: d = xgcd(e, phi)[1]; d
  15417970063428857
❼ sage: mod(d*e, phi)
  1
```

Примечание *Чтобы избежать вывода на несколько страниц, я в листинге 10.1 использую 64-битовый модуль n, но на практике модуль RSA должен быть по меньшей мере 2048-битовым.*

Функция `random_prime()` возвращает случайные простые числа `p` ❶ и `q` ❷, меньшие заданного аргумента. Затем мы перемножаем `p` и `q` и получаем модуль `n` ❸ и значение $\varphi(n)$ в переменной `phi` ❹. Далее

генерируется открытый показатель степени `e` ❺ – случайное простое число, меньшее `phi`; его простота гарантирует, что у `e` будет обратный элемент по модулю `phi`. После этого с помощью функции `xgcd()` генерируется соответствующий закрытый показатель степени *d* ❻. Эта функция, пользуясь обобщенным алгоритмом Евклида, вычисляет по данным числам *a* и *b* такие два числа *s* и *t*, что $as + bt = \mathbf{GCD}(a, b)$ (GCD – наибольший общий делитель, НОД). Наконец, мы проверяем, что $ed \bmod \varphi(n) = 1$ ❼, чтобы удостовериться, что *d* позволит правильно обратить перестановку RSA.

Теперь можно применить перестановку с потайным входом, как показано в листинге 10.2.

Листинг 10.2. Вычисление перестановки с потайным входом в обе стороны

```
❶ sage: x = 1234567
❷ sage: y = power_mod(x, e, n); y
  19048323055755904
❸ sage: power_mod(y, d, n)
  1234567
```

Мы присваиваем *x* целое значение 1234567 ❶, а затем вызываем функцию `power_mod(x, e, n)` возведения в степень по модулю *n* для вычисления *y* ❷. Вычислив $y = x^e \bmod n$, мы вычисляем $y^d \bmod n$ ❸, где *d* – потайной ход, и таким образом возвращаемся к исходному *x*.

Но насколько трудно найти *x*, не зная потайного входа *d*? Противник, умеющий факторизовать большие числа, может взломать RSA, найдя *p* и *q*, а затем $\varphi(n)$, после чего сможет вычислить *d* по *e*. Но это не единственная опасность. Другой риск RSA вытекает из умения противника вычислять *x* по $x^e \bmod n$, т. е. корни степени *e* по модулю *n*, не факторизуя при этом *n*. Оба риска кажутся тесно связанными, хотя мы точно не знаем, эквивалентны ли они.

В предположении, что факторизация – действительно трудная задача, а вычислить корни степени *e* примерно столь же трудно, безопасность RSA зависит от трех факторов: величины *n*, выбора *p* и *q* и способа использования потайного входа. Если *n* слишком мало, то его можно разложить на множители за разумное время и тем самым узнать закрытый ключ. Для безопасности длина *n* должна быть не менее 2048 бит (при этом уровень безопасности составляет приблизительно 90 бит, так что для вскрытия требуется порядка 2^{90} операций), а лучше 4096 бит (уровень безопасности приблизительно 128 бит). Значения *p* и *q* должны быть не связанными между собой простыми числами примерно одинакового размера. Если они слишком малы или слишком близки друг к другу, то определить их по *n* будет проще. Наконец, перестановка с потайным входом не должна использоваться непосредственно для шифрования или подписания – об этом мы поговорим ниже.

Шифрование с помощью RSA

Обычно RSA используется в сочетании с симметричной схемой шифрования, именно с помощью RSA шифруется симметричный ключ, который затем применяется для шифрования сообщения, например шифром AES. Но шифрование сообщения или симметричного ключа с помощью RSA сложнее, чем просто преобразование в число x и вычисление $x^e \bmod n$.

В следующих подразделах я объясню, почему наивное применение перестановки с потайным входом небезопасно и как работает стойкое шифрование на основе RSA.

Взлом RSA-шифрования по учебнику и податливость

Фразой «RSA-шифрование по учебнику» описывают бесхитростную схему шифрования, когда открытый текст содержит только сообщение, которое мы хотим зашифровать. Например, чтобы зашифровать строку *RSA*, мы должны были бы сначала преобразовать ее в число, конкатенировав коды ASCII все трех букв: *R* (байт 52), *S* (байт 53), *A* (байт 41). Получившаяся строка байтов `525341` в десятичном виде равна 5 395 265, и теперь мы могли бы зашифровать ее, вычислив $5\,395\,265^e \bmod n$. Не зная секретного ключа, дешифрировать сообщение было бы невозможно.

Однако RSA-шифрование по учебнику детерминировано: зашифровав один и тот же текст дважды, мы получим один и тот же шифртекст. Это одна проблема, но есть и другая, более серьезная – если даны два шифртекста, зашифрованных по учебнику, $y_1 = x'_1 \bmod n$ и $y_2 = x'_2 \bmod n$, то можно вывести шифртекст $x_1 \times x_2$, перемножив эти шифртексты:

$$y_1 \times y_2 \bmod n = x'_1 \times x'_2 \bmod n = (x_1 \times x_2)' \bmod n.$$

Результат $(x_1 \times x_2)' \bmod n$ – это шифртекст сообщения $x_1 \times x_2 \bmod n$. Таким образом, противник мог бы создать новый правильный шифртекст по двум шифртекстам RSA и тем самым скомпрометировать безопасность шифрования, поскольку сумел бы получить какую-то информацию об исходном сообщении. Говорят, что эта слабость делает RSA-шифрование по учебнику *податливым*. (Разумеется, если вы знаете x_1 и x_2, то тоже можете вычислить $(x_1 \times x_2)' \bmod n$, но знания только y_1 и y_2 не должно быть достаточно для получения шифртекста перемноженных открытых текстов.)

Стойкое RSA-шифрование: OAEP

Чтобы шифртекст RSA был неподатливым, он должен включать данные сообщения, а также *дополнение*, как показано на рис. 10.1. Стандартный способ такого RSA-шифрования – схема оптимального асимметричного шифрования с дополнением (Optimal Asymmetric

Encryption Padding – OAEP). Этот метод, обычно называемый RSA-OAEP, предполагает создание битовой строки такого же размера, как модуль, для чего сообщение дополняется случайными данными перед применением функции RSA.

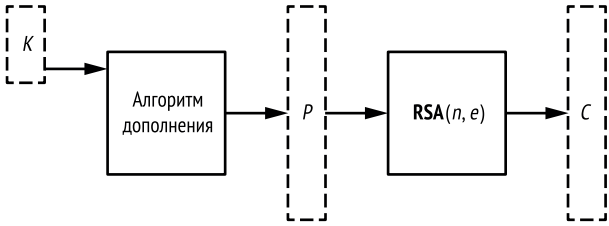

Рис. 10.1. RSA-шифрование симметричного ключа K с использованием (n, e) в качестве открытого ключа

Примечание *OAEP именуется RSAES-OAEP в официальных документах, например в стандарте PKCS#1, выпущенном компанией RSA, и в специальной публикации NIST 800-56B. OAEP является усовершенствованием более раннего метода, который теперь называется PKCS#1 v1.5 и был одним из первых в серии стандартов криптографии с открытым ключом (PKCS), созданных RSA. Он заметно менее безопасен, чем OAEP, но до сих пор используется во многих системах.*

Безопасность OAEP

В OAEP используется генератор псевдослучайных чисел (PRNG), гарантирующий неразличимость и неподатливость шифртекстов, что делает шифрование вероятностным. Доказано, что схема безопасна при условии, что безопасны функция RSA и PRNG и, хотя это менее важно, функции хеширования не слишком слабые. При использовании RSA-шифрования всегда следует применять режим OAEP.

Как работает шифрование в режиме OAEP

Чтобы применить RSA-шифрование в режиме OAEP, необходимо сообщение (обычно симметричный ключ, K), PRNG и две функции хеширования. Для создания шифртекста используется заданный модуль n длиной m байт (т. е. $8m$ бит, так что должно быть меньше n 2^{8m}). Чтобы зашифровать K, формируется *кодированное сообщение* $M = H||00 \ldots 00||01||K$, где H – h-байтовая константа, определяемая схемой OAEP, за которой следует необходимое количество байтов 00 и один байт 01. Сообщение M затем обрабатывается, как описано ниже и изображено на рис. 10.2.

Затем генерируется h-байтовая случайная строка R и выполняется замена $M = M \oplus \mathbf{Hash1}(R)$, где $\mathbf{Hash1}(R)$ имеет такую же длину, как M. Далее мы полагаем $R = R \oplus \mathbf{Hash2}(M)$, где $\mathbf{Hash2}(M)$ имеет такую же длину, как R. Теперь новые значения M и R применяются для формирования m-байтовой строки $P = 00||M||R$ такой же длины, как модуль

n, которую можно преобразовать в целое число, меньшее n. Результатом такого преобразования является число x, которое используется для вычисления функции RSA $x^e \bmod n$ и получения шифртекста.

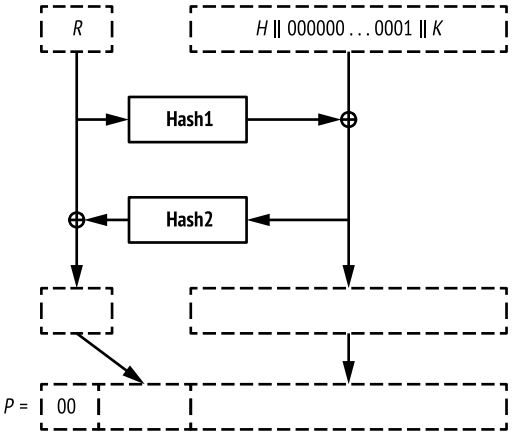

Рис. 10.2. Шифрование симметричного ключа K в режиме RSA-OAEP, где H – фиксированный параметр, а R – случайные биты

Чтобы дешифрировать шифртекст y, мы сначала вычисляем $x = y^d \bmod n$, а затем восстанавливаем значения M и R. Далее вычисляется исходное значение M по формуле $M \oplus \textbf{Hash1}(R \oplus \textbf{Hash2}(M))$. Наконец, мы проверяем, что M имеет вид $H||00 \ldots 00||01||K$, где длина H равна h байт, а после серии байтов 00 находится байт 01.

На практике параметры m и h (длина модуля и длина результата **Hash2** соответственно) обычно равны m = 256 байт (для 2048-битового RSA) и h = 32 (при использовании SHA-256 в качестве **Hash2**). Следовательно, остается $m - h - 1$ = 223 байта для M, из которых до $m - 2h - 2$ = 190 байт доступны для K (член «–2» – это длина разделительного байта 01). Хеш-значение **Hash1** тогда содержит $m - h - 1$ = 223 байта – больше, чем хеш-значение любой из общеупотребительных функций хеширования.

Примечание *Для построения хеша такой необычной длины в стандартах RSA описывается применение метода функций генерирования масок, который создает хеш-функции, возвращающие хеш-значения произвольной длины, по любой хеш-функции.*

Подписание с помощью RSA

Цифровая подпись доказывает, что владелец закрытого ключа действительно подписал некоторое сообщение и что подпись подлинная. Поскольку никто, кроме владельца закрытого ключа, не знает секретного показателя степени d, никто не может вычислить подпись $y = x^d \bmod n$ для некоторого значения x, но любой желающий может про-

верить, что $y^e \bmod n = x$, если знает открытый показатель степени e. Такую верифицированную подпись можно использовать в суде для доказательства того, что владелец закрытого ключа действительно подписал сообщение, – это свойство называется *неотрицаемостью*.

Возникает искушение рассматривать RSA-подписи как процесс, обратный шифрованию, но это не так. Подписание с помощью RSA – не то же самое, что шифрование закрытым ключом. Шифрование обеспечивает конфиденциальность, тогда как цифровая подпись служит для предотвращения подлога. Самый красноречивый пример этого различия – тот факт, что подписание не препятствует разглашению информации сообщения, потому что само сообщение не секретное. Схема, раскрывающая части сообщений, может быть безопасной схемой подписания, ни никак не безопасной схемой шифрования.

Из-за существенных накладных расходов шифрование с открытым ключом может применяться только для обработки коротких сообщений – обычно секретных ключей, а не самих сообщений. Но схема подписания может обрабатывать сообщения произвольной длины благодаря использованию хеш-значений **Hash**(M) в качестве заместителей сообщений; при этом она может быть детерминированной, оставаясь безопасной. Как и в случае RSA-OAEP, в схемах подписания на основе RSA можно пользоваться дополнением, но можно также использовать максимальное место для сообщения, допускаемое модулем RSA.

Взлом RSA-подписей по учебнику

RSA-подписью по учебнику называется метод подписания сообщения x, заключающийся в прямом вычислении $y = x^d \bmod n$, где x – любое число от 0 до $n - 1$. Как и шифрование по учебнику, RSA-подписание по учебнику легко описать и реализовать, но оно уязвимо к нескольким атакам. Одна такая атака сводится к тривиальному подлогу: поскольку $0^d \bmod n = 0$, $1^d \bmod n = 1$ и $(n - 1)^d \bmod n = n - 1$ независимо от значения закрытого ключа d, противник может подделать подписи 0, 1 и $n - 1$, не зная d.

Больше опасений вызывает *атака с ослеплением*. Например, предположим, что мы хотим получить подпись третьей стороны для некоторого разоблачающего сообщения M, которое она заведомо не подписала бы, зная, что в нем содержится. Для организации атаки можно было бы сначала найти такое значение R, что $R^eM \bmod n$ является сообщением, которое жертва готова была бы подписать. Затем можно было бы убедить жертву подписать это сообщение и показать нам свою подпись, равную $S = (R^eM)^d \bmod n$, т. е. сообщению, возведенному в степень d. Теперь, зная эту подпись, мы можем вывести подпись M, а именно M^d, выполнив тривиальные вычисления.

Работает это так: поскольку S можно записать в виде $(R^eM)^d = R^{ed}M^d$, а $R^{ed} = R$ (по определению), имеем $S = (R^eM)^d = RM^d$. Чтобы получить M^d, мы просто делим S на R и получаем подпись:

$$S/R = RM^d/R = M^d.$$

Как видим, это практичная и действенная атака.

Стандарт цифровой подписи PSS

Стандарт *вероятностной схемы подписания* (Probabilistic Signature Scheme – *PSS*) играет для RSA-подписей ту же роль, что OAEP для RSA-шифрования. Он был спроектирован, чтобы сделать подписание сообщения более безопасным за счет добавления данных дополнения.

Как показано на рис. 10.3, PSS объединяет сообщение, меньшее по длине, чем модуль, со случайными и фиксированными битами, а затем применяет RSA к результату этой процедуры дополнения.

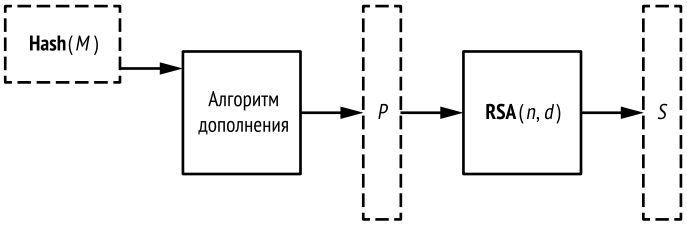

Рис. 10.3. RSA-подписание сообщения M по стандарту PSS, где (n, d) – закрытый ключ

Как и все схемы подписания с открытым ключом, PSS применяется к хешу сообщения, а не к самому сообщению. Подписание **Hash**(*M*) безопасно при условии стойкости функции хеширования к коллизиям. Одно из преимуществ PSS – возможность его использования для подписания сообщений любой длины, потому что после хеширования сообщения мы получаем хеш-значение одной и той же длины, какой бы ни была длина оригинального сообщения. Типичная длина хеш-значения – 256 бит, если в качестве функции хеширования используется SHA-256.

Почему бы не подписывать сообщение, просто применяя OAEP к **Hash**(*M*)? К сожалению, так нельзя. Хотя режим OAEP и похож на PSS, его безопасность доказана только для шифрования, но не для подписания.

Как и OAEP, PSS требует PRNG и двух функций хеширования. Одна, **Hash1**, возвращает типичное хеш-значение длиной *h* байт, например SHA-256. Другая, **Hash2**, формирует широкий выходной хеш, как **Hash2** в OAEP.

Процедура PSS подписания сообщения *M* работает следующим образом (где *h* – длина выхода **Hash1**).

1. Случайно выбрать *r*-байтовую строку *R*, применив PRNG.
2. Сформировать кодированное сообщение *M′* = 0000000000000000 || **Hash1**(*M*) || *R* длиной *h* + *r* + 8 байт (содержащее восемь нулевых байт в начале).
3. Вычислить *h*-байтовую строку *H* = **Hash1**(*M’*).

4. Положить $L = 00 \ldots 00 || 01 || R$ – последовательность байтов 00, за которой следует байт 01 и R, а байтов 00 столько, чтобы длина L составляла $m - h - 1$ байт (ширина m модуля в байтах за вычетом длины хеша и единицы).
5. Положить $L = L \oplus$ **Hash2**(H), заменив предыдущее значение L новым.
6. Преобразовать m-байтовую строку $P = L || H || \text{BC}$ в число x, меньшее n. Здесь байт BC – фиксированное значение, дописанное после H.
7. Для полученного значения x вычислить функцию RSA $x^d \bmod n$, она и будет подписью.

Чтобы верифицировать подпись заданного сообщения M, следует вычислить **Hash1**(M) и воспользоваться открытым показателем степени e для выделения L и H, а затем M' из подписи, проверяя на каждом шаге правильность дополнения.

На практике длина случайной строки R (называемой *солью* в стандарте RSA-PSS) обычно такая же, как длина хеш-значения. Например, если используется $n = 2048$ бит и SHA-256 в качестве функции хеширования, то длина L равна $m - h - 1 = 256 - 32 - 1 = 223$ байта, а длина случайной строки R обычно равна 32 байта.

Как и OAEP, схема PSS доказуемо безопасна, стандартизована и широко применяется. И так же, как OAEP, она выглядит беспричинно усложненной, из-за чего возможны ошибки реализации и нерассмотренные редкие случаи. Но, в отличие от RSA-шифрования, есть способ избежать этой излишней сложности, применив схему подписания, в которой даже PRNG не нужен, и тем самым уменьшить риск появления небезопасных подписей, вызванный небезопасным PRNG. Читайте дальше.

Подписи на основе полного хеша домена

Полный хеш домена (Full Domain Hash – FDH) – простейшая мыслимая схема цифровой подписи. Для ее реализации мы просто преобразуем байтовую строку **Hash**(M) в число x и создаем подпись $y = x^d \bmod n$, как показано на рис. 10.4.

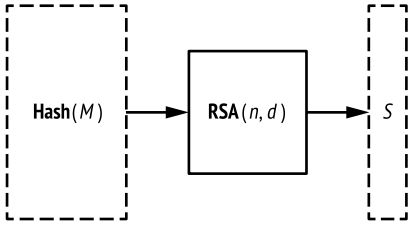

Рис. 10.4. RSA-подписание сообщения методом полного хеша домена

Верификация подписи также не вызывает трудностей. Имея подпись – число y, мы вычисляем $x = y^e \bmod n$ и сравниваем результат

с **Hash**(*M*). Скучно как просто, детерминировано, но при всем при том безопасно. Так зачем же нужна сложность PSS?

Основная причина – в том, что стандарт PSS был опубликован *после* FDH, в 1996 году, и для него имеется доказательство безопасности, которое вселяет больше уверенности, чем FDH. Точнее, это доказательство дает чуть более сильные гарантии безопасности, чем доказательство для FDH, чему немало способствует использование случайности.

Наличие более сильных теоретических гарантий – главная причина, по которой криптографы предпочитают PSS, а не FDH, но большинство современных приложений, использующих PSS, могли бы перейти на FDH практически без потери безопасности. Однако в некоторых контекстах существует веская причина использовать именно PSS, потому что случайность защищает от некоторых атак на реализацию, например атак на недочеты, которые мы обсудим в разделе «Какие возможны проблемы».

Реализации RSA

Я искренне надеюсь, что вам никогда не придется реализовывать RSA с нуля. Если вас попросят это сделать, бегите как можно быстрее и задайтесь вопросом, не безумен ли человек, обратившийся с такой просьбой. Криптографам и инженерам потребовалось несколько десятилетий, чтобы разработать реализации RSA, которые были бы быстрыми, достаточно безопасными и, хочется надеяться, не содержали деструктивных ошибок. Поэтому заново изобретать RSA точно не нужно. Даже при наличии всей документации решение этой грандиозной задачи заняло бы несколько месяцев.

Обычно при реализации RSA используется библиотека или API, предлагающие все необходимые функции для выполнения операций RSA. Например, в пакете `crypto` для языка Go имеется следующая функция (*https://www.golang.org/src/crypto/rsa/rsa.go*):

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte,
label []byte) (out []byte, err error)
```

Функция `EncryptOAEP()` принимает хеш-функцию, PRNG, открытый ключ, сообщение и метку (факультативный параметр OAEP) и возвращает подпись и код ошибки. Она вызывает `encrypt()` для вычисления функции RSA, передавая ей дополненные данные, как показано в листинге 10.3.

Листинг 10.3. Реализация базовой функции RSA-шифрования в криптографической библиотеке для языка Go

```
func encrypt(c *big.Int, pub *PublicKey, m *big.Int) *big.Int {
    e := big.NewInt(int64(pub.E))
```

```
    c.Exp(m, e, pub.N)
    return c
}
```

Основная операция, показанная в листинге 10.3, `c.Exp(m, e, pub.N)`, возводит сообщение `m` в степень `e` по модулю `pub.N` и присваивает результат переменной `c`.

Если вы все-таки решили реализовывать RSA, а не пользоваться готовой библиотечной функцией, то хотя бы опирайтесь на имеющуюся библиотеку для работы с *большими числами*, содержащую набор функций и типов, позволяющих определять и вычислять арифметические операции над большими числами длиной тысячи бит. Например, можно воспользоваться арифметической библиотекой GNU Multiple Precision (GMP), написанной на C, или пакетом Go `big`. (Поверьте мне, лучше не стоит реализовывать арифметику больших чисел самостоятельно.)

Но даже если вы пользуетесь библиотечной функцией при реализации RSA, разберитесь, как она работает, чтобы понять меру риска.

Алгоритм быстрого возведения в степень

Операция $x^e \bmod n$ называется *возведением в степень*. Будучи наивно реализована, эта операция может оказаться чрезвычайно медленной при работе с большими числами, в частности в RSA. Но как реализовать ее эффективно?

При наивном вычислении $x^e \bmod n$ потребуется $e - 1$ умножений, как показано в алгоритме на псевдокоде в листинге 10.4.

Листинг 10.4. Псевдокод наивного алгоритма возведения в степень

```
expModNaive(x, e, n) {
    y = x
    for i = 1 to e - 1 {
        y = y * x mod n
    }
    return y
}
```

Этот алгоритм прост, но абсолютно не эффективен. Тот же результат можно получить экспоненциально быстрее, если не перемножать числа, а возводить их в квадрат, пока не дойдем до нужного показателя степени. Это семейство методов называется алгоритмами *быстрого*, или *двоичного*, *возведения в степень*.

Например, пусть требуется вычислить $3^{65537} \bmod 36567232109354321$ (число 65 537 – открытый показатель степени, используемый в большинстве реализаций RSA). Можно было бы умножить 3 на себя 65 536 раз, или заметить, что $65\,537 = 2^{16} + 1$, и выполнить серию операций возведения в квадрат. Именно инициализируем переменную $y = 3$, а затем выполним такие операции возведения в квадрат (y^2):

1. Положить $y = y^2 \bmod n$ (теперь $y = 3^2 \bmod n$).
2. Положить $y = y^2 \bmod n$ (теперь $y = (3^2)^2 = 3^4 \bmod n$).
3. Положить $y = y^2 \bmod n$ (теперь $y = (3^4)^2 = 3^8 \bmod n$).
4. Положить $y = y^2 \bmod n$ (теперь $y = (3^8)^2 = 3^{16} \bmod n$).
5. Положить $y = y^2 \bmod n$ (теперь $y = 3^{16})^2 = 3^{32} \bmod n$).

И так далее, пока не дойдем до $y = 3^{65536}$, для чего понадобится 16 возведений в квадрат.

Для получения окончательного результата возвращаем $3 \times y \bmod n = 3^{65537} \bmod n = 26\,652\,909\,283\,612\,267$. Как видим, для вычисления результата понадобилось всего 17 умножений вместо 65 536.

В общем случае для быстрого возведения в квадрат нужно просматривать биты показателя степени слева направо, возводя текущий результат в квадрат, если очередной бит равен 0, и умножая его на исходное число, если бит равен 0. В рассмотренном выше примере показатель степени равен 65 537, или 10000000000000001 в двоичном виде, поэтому мы каждый раз возводили y в квадрат, кроме первого и последнего шагов, на котором умножали на исходное число 3.

В листинге 10.5 показан общий псевдокод алгоритма быстрого вычисления $x^e \bmod n$ в случае, когда показатель степени e содержит биты $e_{m-1}e_{m-2} \ldots e_1e_0$, где e_0 – младший бит.

Листинг 10.5. Псевдокод алгоритма быстрого возведения в степень

```
expMod(x, e, n) {
    y = x
    for i = m - 1 to 0 {
        y = y * y mod n
        if e_i == 1 then
        y = y * x mod n
    }
    return y
}
```

Алгоритм `expMod()` в листинге 10.5 работает за время $O(m)$, а наивный алгоритм за время $O(2^m)$, где m – длина показателя степени в битах. Здесь $O()$ обозначает асимптотическую сложность (см. главу 9).

В реальных системах часто реализуются варианты этого простейшего метода быстрого возведения в степень. В одном из них, методе *скользящего окна*, рассматриваются блоки битов, а не отдельные биты. Например, см. функцию `expNN()` на языке Go, исходный код которой доступен по адресу *https://golang.org/src/math/big/nat.go*.

Насколько безопасны алгоритмы быстрого возведения в степень? К сожалению, различные приемы ускорения процесса часто ведут к повышенной уязвимости к некоторым атакам. Посмотрим, какие могут возникнуть проблемы.

Слабость этих алгоритмов обусловлена тем, что операции возведения в степень сильно зависят от показателя степени. Операция `if`

в листинге 10.5 выбирает разные ветви в зависимости от значения очередного бита показателя степени. Если бит равен 1, то итерация цикла `for` будет медленнее, чем для 0, и противник, измеряющий время выполнения операции RSA, может воспользоваться этим различием, чтобы восстановить закрытый показатель степени. Это так называемая атака с хронометражем. Атаки на оборудование могут различить единичные и нулевые биты, отслеживая энергопотребление устройства и наблюдая за тем, на каких итерациях выполняется дополнительное умножение, – это поможет определить единичные биты закрытого показателя степени.

Лишь в немногих криптографических библиотеках реализована эффективная защита от атак с хронометражем, не говоря уже об атаках с измерением энергопотребления.

Выбор малых показателей степени для ускорения операций с открытым ключом

Поскольку вычисление RSA в значительной степени сводится к возведению в степень, его производительность зависит от величины показателя степени. Чем меньше показатель, тем меньше требуется умножений, поэтому вычисление степени занимает гораздо меньше времени.

Открытый показатель степени e в принципе может быть любым числом от 3 до $\varphi(n) - 1$, лишь бы e и $\varphi(n)$ были взаимно простыми. Но на практике встречаются только небольшие значения e, чаще всего $e = 65\,537$ – это связано с желанием ускорить шифрование и верификацию подписи. Например, Microsoft Windows CryptoAPI поддерживает только открытые показатели степени, которые можно записать 32-битовым целым числом. Чем больше e, тем дольше вычисляется $x^e \bmod n$.

В отличие от открытого, закрытый показатель степени d должен быть примерно такого же размера, как n, в результате чего дешифрирование оказывается гораздо медленнее шифрования, а подписание гораздо медленнее верификации. Действительно, поскольку d – секрет, он должен быть непредсказуемым, так что ограничиться малыми значениями не получится. Например, если e фиксировано и равно 65 537, то соответствующее d обычно выбирается по порядку величин таким же, как модуль n, т. е. близко к 2^{2048}, если длина n равна 2048 бит.

В разделе «Алгоритм быстрого возведения в степень» выше мы видели, что возведение числа в степень 65 537 требует всего 17 умножений, тогда как возведение в степень, записываемую 2048-битовым числом, потребует порядка 3000 умножений.

Один из способов определить фактическую скорость RSA – воспользоваться пакетом OpenSSL. Так, в листинге 10.6 приведены результаты операций 512-, 1024-, 2048- и 4096-битового RSA на моем компьютере MacBook с процессором Intel Core i5-5257U с тактовой частотой 2.7 ГГц.

Листинг 10.6. Ориентиры для времени операций RSA, полученные с помощью пакета OpenSSL

```
$ openssl speed rsa512 rsa1024 rsa2048 rsa4096
Doing 512 bit private rsa's for 10s: 161476 512 bit private RSA's in 9.59s
Doing 512 bit public rsa's for 10s: 1875805 512 bit public RSA's in 9.68s
Doing 1024 bit private rsa's for 10s: 51500 1024 bit private RSA's in 8.97s
Doing 1024 bit public rsa's for 10s: 715835 1024 bit public RSA's in 8.45s
Doing 2048 bit private rsa's for 10s: 13111 2048 bit private RSA's in 9.65s
Doing 2048 bit public rsa's for 10s: 288772 2048 bit public RSA's in 9.68s
Doing 4096 bit private rsa's for 10s: 1273 4096 bit private RSA's in 9.71s
Doing 4096 bit public rsa's for 10s: 63987 4096 bit public RSA's in 8.50s
OpenSSL 1.0.2g 1 Mar 2016
--опущено--
                  sign    verify  sign/s verify/s
rsa  512 bits 0.000059s 0.000005s 16838.0 193781.5
rsa 1024 bits 0.000174s 0.000012s  5741.4  84714.2
rsa 2048 bits 0.000736s 0.000034s  1358.7  29831.8
rsa 4096 bits 0.007628s 0.000133s   131.1   7527.9
```

Насколько верификация подписи медленнее ее создания? Чтобы получить представление, можно вычислить отношение одного к другому. Эталонные цифры в листинге 10.6 показывают, что отношение скорости верификации к скорости создания подписи равно приблизительно 11.51, 14.75, 21.96 и 57.42 для модулей размера 512, 1024, 2048 и 4096 бит соответственно. Разрыв растет с увеличением модуля, потому что количество умножений в операциях с e остается постоянным (например, 17 для $e = 65\,537$), а в операциях с закрытым ключом требуется больше умножений, поскольку d растет вместе с модулем.

Но если малые показатели степени – так замечательно, то почему используется 65 537, а не, скажем, 3? Было бы лучше (и быстрее) использовать 3 в качестве открытого показателя степени при реализации таких безопасных схем RSA, как OAEP, PSS или FDH. Но криптографы так не делают, потому что при $e = 3$ некоторые менее безопасные схемы становятся уязвимы для математических атак. Число 65 537 достаточно велико, чтобы избежать таких *атак на малый показатель степени*, и единичный бит в нем находится только в одной позиции, что уменьшает время вычислений. Кроме того, для математиков число 65 537 несет особый смысл: это четвертое число Ферма, т. е. имеет вид

$$2^{(2n)} + 1,$$

где $n = 4$, но это всего лишь курьез, не имеющий значения для криптографии.

Китайская теорема об остатках

Самый распространенный прием, ускоряющий дешифрирование и генерирование подписи (т. е. вычисление $y^d \bmod n$), – *китайская теорема об остатках*. Она ускоряет RSA примерно в четыре раза.

Китайская теорема об остатках позволяет ускорить дешифрирование за счет вычисления двух возведений степени, по модулям p и q, а не по модулю n. Поскольку p и q гораздо меньше n, проще выполнить два «малых» возведения в степень, чем одно «большое».

Китайская теорема об остатках не имеет прямого отношения к RSA. Это общий арифметический результат; в простейшем виде она утверждает, что если $n = n_1 n_2 n_3 \ldots$, где n_i – попарно взаимно простые числа (т. е. $\mathbf{GCD}(n_i, n_j) = 1$ для любых различных i и j), то значение $x \bmod n$ можно вычислить, зная значения $x \bmod n_1$, $x \bmod n_2$, $x \bmod n_3$, …. Например, пусть число $n = 1155$, оно разлагается в произведение простых множителей $3 \times 5 \times 7 \times 11$. Мы хотим найти такое x, что $x \bmod 3 = 2$, $x \bmod 5 = 1$, $x \bmod 7 = 6$ и $x \bmod 11 = 8$ (числа 2, 1, 6, 8 выбраны произвольно).

Чтобы найти x, воспользовавшись китайской теоремой об остатках, вычислим сумму $P(n_1) + P(n_2) + \ldots$, где $P(n_i)$ определено следующим образом:

$$P(n_i) = (x \bmod n_i) \times n/n_i \times (1/(n/n_i) \bmod n_i) \bmod n.$$

Заметим, что второй член, n/n_i, равен произведению всех множителей, кроме n_i.

Чтобы применить эту формулу к нашему примеру и восстановить $x \bmod 1155$, выберем произвольные значения 2, 1, 6, 8, вычислим $P(3)$, $P(5)$, $P(7)$ и $P(11)$, а затем сложим и получим:

$$[2 \times 385 \times (1/385 \bmod 3) + 1 \times 231 \times (1/231 \bmod 5) + 6 \times 165 \times (1/165 \bmod 7) + 8 \times 105 \times (1/105) \bmod 11] \bmod n.$$

Здесь я просто применил данное выше определение $P(n_i)$. (Математическое обоснование способа вычисления каждого числа несложно, но я не буду на нем останавливаться.) Это выражение можно свести к $[770 + 231 + 1980 + 1680] \bmod n = 41$, и я действительно выбрал 41, так что мы получили правильный результат.

Применить китайскую теорему об остатках к RSA проще, чем в рассмотренном примере, потому что для каждого n есть всего два множителя (p и q). Получив подлежащий дешифрированию шифртекст y, мы не будем вычислять $y^d \bmod n$, а применим китайскую теорему об остатках и вычислим $x_p = y^s \bmod p$, где $s = d \bmod (p - 1)$, и $x_q = y^t \bmod q$, где $t = d \bmod (q - 1)$. Затем объединим оба выражения и вычислим x по формуле

$$x = x_p \times q \times (1/q \bmod p) + x_q \times p \times (1/p \bmod q) \bmod n.$$

Вот и все. Это быстрее, чем метод двоичного возведения в квадрат, потому что умножения выполняются по модулям p и q, которые в два раза короче n.

Примечание *В последней операции числа $q \times (1/q \bmod p)$ и $p \times (1/p \bmod q)$ можно вычислить заранее, а значит, для нахождения x нужно будет выполнить только два умножения и сложение по модулю n.*

К сожалению, эта техника сопряжена с угрозой безопасности, которую мы обсудим ниже.

Какие возможны проблемы

Схема RSA красива, но еще красивее серия атак, работающих либо потому, что реализация дает утечку информации о внутреннем механизме (или ее можно заставить дать такую утечку), либо потому, что RSA небезопасно используется. Ниже я опишу два классических примера атак такого рода.

Атака Bellcore на RSA-CRT

Атака Bellcore – одна из самых важных в истории RSA. Открытая в 1996 году, она стоит особняком, потому что эксплуатировала уязвимость RSA к *внесению ошибок* – атак, которые заставляют часть алгоритма вести себя неправильно и давать некорректные результаты. Например, аппаратные схемы или встраиваемые системы можно подвергнуть временным возмущениям, внезапно изменив входное напряжение или направив лазерный импульс на тщательно выбранную часть микросхемы. Затем противник может воспользоваться возникшими ошибками в работе алгоритма, наблюдая влияние на конечный результат. Например, сравнение правильного результата с ошибочными может дать информацию о внутренних значениях алгоритма, в т. ч. секретных.

Атака Bellcore относится именно к такому классу атак. Она эффективна против детерминированных схем RSA-подписи, в которых применяется китайская теорема об остатках, т. е. против FDH, но не против вероятностной схемы PSS.

Чтобы понять, как работает атака Bellcore, вспомним, что, в силу китайской теоремы об остатках, результат, равный $x^d \bmod n$, получается вычислением следующего выражения, в котором $x_p = y^s \bmod p$ и $x_q = y^t \bmod q$:

$$x = x_p \times q \times (1/q \bmod p) + x_q \times p \times (1/p \bmod q) \bmod n.$$

Теперь предположим, что противник вносит ошибку в вычисление x_q, так что в итоге получается неправильное значение, отличающееся от настоящего x_q. Обозначим это неправильное значение x'_q, а конечный результат x'. Тогда противник может вычесть неправильную подпись x' из правильной подписи x, чтобы факторизовать n:

$$x - x' = (x_q - x'_q) \times p \times (1/p \bmod q) \bmod n.$$

Таким образом, значение $x - x'$ кратно p, т. е. p является делителем $x - x'$. Поскольку p также является делителем n, то наибольшим общим делителем n и $x - x'$ будет p, $\mathbf{GCD}(x - x', n) = p$. Тогда можно вычислить $q = n/p$ и d и, следовательно, полностью взломать RSA-подписи.

Вариант этой атаки работает, когда правильная подпись неизвестна, но известно, что сообщение подписано. Существует похожая атака внесением ошибки на значение модуля, а не на вычисление значений в китайской теореме об остатках, но здесь я не стану вдаваться в ее детали.

Разделение закрытых показателей степени или модулей

Теперь я покажу, почему ваш открытый ключ не должен иметь такой же модуль n, как чей-то еще.

Различные закрытые ключи, принадлежащие разным системам или физическим лицам, очевидно, должны иметь разные закрытые показатели степени d, даже если у ключей разные модули, иначе вы могли бы пробовать свое значение d для дешифрирования чужих сообщений, пока не наткнетесь на сообщение с таким же d. По тем же причинам в разных парах ключей должны быть разные значения n, даже если значения d различны, потому что p и q обычно являются частями закрытого ключа. Следовательно, если мы разделяем одно и то же n, а значит, одинаковые p и q, то я смогу вычислить ваш закрытый ключ по вашему открытому ключу e, используя p и q.

А что, если мой закрытый ключ – это просто пара (n, d_1), ваш закрытый ключ – пара (n, d_2), а ваш открытый ключ – пара (n, e_2)? Допустим, что я знаю n, но не знаю p и q, поэтому не могу напрямую вычислить ваш закрытый показатель степени d_2 по вашему открытому показателю степени e_2. Как бы вы вычислили p и q, зная только закрытый показатель степени d? Решение элегантное, хотя опирается на некоторые математические факты.

Вспомним, что d и e связаны соотношением $ed = k\varphi(n) + 1$, где $\varphi(n)$ – секрет, знание которого помогло бы узнать p и q непосредственно. Мы не знаем ни k, ни $\varphi(n)$, но можем вычислить $k\varphi(n) = ed - 1$.

Что можно сделать с этим значением $k\varphi(n)$? Во-первых, согласно *теореме Эйлера*, мы знаем, что для любого числа a, взаимно простого с n, $a^{\varphi(n)} = 1 \bmod n$. Поэтому по модулю n имеет место равенство:

$$a^{k\varphi(n)} = (a^{\varphi(n)})^k = 1^k = 1.$$

Во-вторых, поскольку $k\varphi(n)$ – четное число, мы можем записать его в виде $2^s t$ для некоторых s и t. То есть мы сможем записать равенство $a^{k\varphi(n)} = 1 \bmod n$ в виде $x^2 = 1 \bmod n$ для некоторого x, которое легко вычисляется по $k\varphi(n)$. Такое x называется *корнем из единицы*.

Ключевое наблюдение заключается в том, что равенство $x^2 = 1 \bmod n$ равносильно тому, что $x^2 - 1 = (x - 1)(x + 1)$ делится на n. Иными словами, либо $x - 1$, либо $x + 1$ должны иметь общий множитель с n, что может дать нам факторизацию n.

В листинге 10.7 показана реализация этого метода на Python, в которой, чтобы без труда найти множители p и q по n и d, мы использовали небольшие 64-битовые числа.

Листинг 10.7. Python-программа, вычисляющая простые множители p и q по закрытому показателю степени d

```
from math import gcd

n = 36567232109354321
e = 13771927877214701
d = 15417970063428857

❶ kphi = d*e - 1
  t = kphi

❷ while t % 2 == 0:
      t = divmod(t, 2)[0]

❸ a = 2
  while a < 100:
    ❹ k = t
      while k < kphi:
          x = pow(a, k, n)
      ❺   if x ! = 1 and x ! = (n - 1) and pow(x, 2, n) == 1:
          ❻ p = gcd(x - 1, n)
              break
          k = k*2
      a = a + 2

  q = n//p
❼ assert (p*q) == n
  print('p = ', p)
  print('q = ', q)
```

Эта программа находит $k\varphi(n)$ по e и d ❶, отыскав число t такое, что $k\varphi(n) = 2^s t$ для некоторого s ❷. Затем она ищет a и k такие, что $(a^k)^2 = 1 \bmod n$ ❸, используя t как начальное значение k ❹. Как только это условие будет выполнено ❺, мы нашли решение. Затем программа находит множитель p ❻ и проверяет ❼, что pq равно n. Наконец, печатаются найденные p и q:

```
p = 2046223079
q = 17870599
```

Программа правильно нашла оба множителя.

Для дополнительного чтения

RSA заслуживает отдельной книги. Я вынужден был опустить много важных и интересных вещей, например атаку Блейхенбахера с оракулом дополнения на предшественника OAEP (стандарт PKCS#1 v1.5), по духу напоминающую атаку с оракулом дополнения на блочные шифры (см. главу 4). Существует также атака Винера на RSA с малыми

закрытыми показателями степени и атака методом Копперсмита на RSA с малыми показателями степени, которые потенциально могут иметь небезопасное дополнение.

Для знакомства с результатами, относящимися к атакам по побочным каналам и защите от них, см. материалы Симпозиума по криптографическому оборудованию и встраиваемым системам (CHES), который проводится с 1999 года (*http://www.chesworkshop.org/*). При написании этой главы очень полезным оказался обзор Boneh «Twenty Years of Attacks on the RSA Cryptosystem», в котором описываются и объясняются самые важные атаки на RSA. Конкретно об атаках с хронометражем см. статью Brumley and Boneh «Remote Timing Attacks Are Practical», которую просто необходимо прочитать ради изложенных в ней экспериментальных фактов и их анализа. Об атаках с внесением ошибок см. полную версию статьи об атаке Bellcore: Boneh, DeMillo, and Lipton «On the Importance of Eliminating Errors in Cryptographic Computations».

Лучший способ изучить, как работает RSA, пусть временами мучительный и чреватый разочарованиями, – познакомиться с исходным кодом какой-нибудь широко распространенной реализации. Например, можете взять код RSA и тесно связанной с ним библиотеки для арифметических операций с большими числами из OpenSSL, или NSS (библиотека, используемая в браузере Mozilla Firefox), или Crypto++, или из любого другого популярного ПО и посмотреть, как реализованы арифметические операции и их защита от атак с хронометражем и внесением ошибок.

11

ПРОТОКОЛ ДИФФИ–ХЕЛЛМАНА

В ноябре 1976 года исследователи из Стэнфордского университета Уитфилд Диффи и Мартин Хеллман опубликовали статью под названием «New Directions in Cryptography», которая произвела революцию, навсегда изменившую лицо криптографии. В этой статье было введено понятие шифрования и подписи с открытым ключом, хотя они не предложили ни одной из этих схем. А предложили они *схему распределения открытых ключей* – протокол, позволяющий двум сторонам выработать общий секрет, обмениваясь информацией по открытым каналам, которые можно прослушать. Теперь этот протокол носит имя *Диффи–Хеллмана* (*DH*).

До появления протокола Диффи–Хеллмана для распространения общего секрета были необходимы утомительные процедуры, например физический обмен запечатанными конвертами. После того как

обе стороны выработали общее секретное значение по протоколу DH, этим секретом можно воспользоваться для установления *безопасного канала*, преобразовав секрет в один или несколько симметричных ключей, с помощью которых шифруется и аутентифицируется последующий обмен данными. Поэтому протокол DH и его варианты называются протоколами совместной выработки ключа.

В первой части этой главы я в общих чертах объясню математические основания протокола Диффи–Хеллмана, включая вычислительные задачи, на которые он опирается. Затем, во второй части, я опишу различные версии протокола Диффи–Хеллмана, применяемые для создания безопасных каналов. Наконец, поскольку схемы DH безопасны только при правильно выбранных параметрах, я завершу главу рассмотрением сценариев, в которых этот протокол может дать сбой.

Примечание *В 2015 году Диффи и Хеллман получили престижную премию Тьюринга за изобретение криптографии с открытым ключом и цифровых подписей, но есть и другие, чей вклад следует отметить. В 1974 году, за два года до основополагающей статьи Диффи и Хеллмана, Ральф Меркл впервые предложил идею криптографии с открытым ключом, представив то, что теперь известно под названием «головоломки Меркла». Примерно тогда же исследователи из Управления правительственной связи, британского эквивалента АНБ, также открыли принципы, положенные в основу RSA и протокола выработки ключей Диффи–Хеллмана, хотя этот факт был рассекречен лишь спустя несколько десятилетий.*

Функция Диффи–Хеллмана

Чтобы разобраться в протоколах совместной выработки ключей DH, нужно сначала понимать их базовую операцию, *функцию Диффи–Хеллмана*. Обычно эта функция работает с группами, обозначаемыми $\mathbf{Z}_p^*$. Напомним (см. главу 9), что эти группы образованы ненулевыми целыми числами по модулю простого числа p. Другой открытый параметр – *основание*, g. Все арифметические операции выполняются по модулю p.

В функции DH участвуют два закрытых значения, a и b, случайно выбранных двумя сообщающимися сторонами из группы $\mathbf{Z}_p^*$. С a ассоциировано открытое значение $A = g^a \bmod p$, т. е. g, возведенное в степень a по модулю p. Это значение отправляется другой стороне в сообщении, доступном для прослушивания. С b ассоциировано открытое значение $B = g^b \bmod p$, которое отправляется владельцу a тоже в виде сообщения, допускающего прослушивание.

Работа DH основана на комбинировании любого открытого значения с другим закрытым значением, при этом результат в обоих случаях одинаков: $A^b = (g^a)^b = g^{ab}$ и $B^a = (g^b)^a = g^{ba} = g^{ab}$. Результат, g^{ab}, –

разделяемый секрет, он передается *функции формирования ключа* (key derivation function – KDF), чтобы та сгенерировала один или несколько разделяемых симметричных ключей. KDF – это разновидность хеш-функции, которая возвращает кажущуюся случайной строку, размер которой равен желаемой длине ключа.

Вот, собственно, и все. Как и многие другие научные открытия (всемирное тяготение, теория относительности, квантовые вычисления или RSA), протокол Диффи–Хеллмана задним числом кажется на удивление простым.

Но простота DH может быть обманчива. Прежде всего он работает не с любым простым p или основанием g. Например, при некоторых значениях g разделяемый секрет g^{ab} ограничен небольшим подмножеством возможных значений, тогда как мы ожидаем, что их будет столько же, сколько элементов в группе $\mathbf{Z}_p^*$. Для обеспечения наивысшей безопасности простое число p нужно выбирать так, чтобы $(p-1)/2$ тоже было простым. Такое *безопасное простое число* гарантирует, что у группы не будет небольших подгрупп, что облегчило бы взлом DH. Если p безопасно, то DH может работать с $g = 2$, что немного ускоряет вычисления. Но для генерирования безопасного простого числа p нужно больше времени, чем для генерирования случайного простого числа.

Например, команда `dhparam` из пакета OpenSSL генерирует только безопасные параметры DH, но встроенные в алгоритм проверки заметно увеличивают время выполнения, как показано в листинге 11.1.

Листинг 11.1. Измерение времени генерирования параметров 2048-битового протокола Диффи–Хеллмана с помощью пакета OpenSSL

```
$ time openssl dhparam 2048
Generating DH parameters, 2048 bit long safe prime, generator 2
This is going to take a long time
--опущено--
-----BEGIN DH PARAMETERS-----
MIIBCAKCAQEAoSIbyA9e844q7V89rcoEV8vd/l2svwhIIjG9EPwWWr7FkfYhYkU9
fRNttmilGCTfxc9EDf+4dzw+AbRBc6oOL9gxUoPnOd1/G/YDYgyplF5M3xeswqea
SD+B7628pWTaCZGKZham7vmiN8azGeaYAucckTkjVWceHVIVXe5fvU74k7+C2wKk
iiyMFm8th2zm9W/shiKNV2+SsHtD6r3ZC2/hfu7XdOI4iT6ise83YicU/cRaDmK6
zgBKn3SlCjwL4M3+m1J+Vh0UFz/nWTJ1IWAVC+aoLK8upqRgApOgHkVqzP/CgwBw
XAOE8ncQqroJ0mUSB5eLqfpAvyBWpkrwQwIBAg==
-----END DH PARAMETERS-----
openssl dhparam 2048 154.53s user 0.86s system 99% cpu 2:36.85 total
```

Как видно, для генерирования параметров понадобилось 154.53 секунды. Для сравнения в листинге 11.2 показано, сколько в той же самой системе ушло времени на генерирование параметров RSA такого же размера (двух простых чисел p и q, размер каждого из которых равен примерно половине размера p).

Листинг 11.2. Измерение времени генерирования параметров 2048-битового RSA

```
$ time openssl genrsa 2048
Generating RSA private key, 2048 bit long modulus
.............................................+++
.....................................................+++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
--опущено--
-----END RSA PRIVATE KEY-----
openssl genrsa 2048 0.16s user 0.01s system 95% cpu 0.171 total
```

Генерирование параметров DH заняло примерно в 1000 раз больше времени, чем параметров RSA с таким же уровнем безопасности, в основном из-за дополнительных ограничений на простое число для протокола DH.

Проблемы протоколов Диффи–Хеллмана

Безопасность протоколов DH зависит от трудности вычислительных задач, особенно задачи о дискретном логарифме (DLP), описанной в главе 9. Очевидно, что DH можно взломать, восстановив закрытое значение a по открытому значению g^a, а это и сводится к решению DLP. Но при использовании DH для вычисления разделяемых секретов нас волнует не только задача о дискретном логарифме, но и две специфичные для DH проблемы, описанные ниже.

Вычислительная задача Диффи–Хеллмана

Вычислительная задача Диффи–Хеллмана (computational Diffie–Hellman – CDH) – это задача вычисления разделяемого секрета g^{ab}, если известны только открытые значения g^a и g^b, но не секретные значения a или b. Побудительный мотив понятен – гарантировать, что даже если противник подслушал g^a и g^b, он не сможет определить разделяемый секрет g^{ab}.

Если можно решить DLP, то можно решить и CDH; проще говоря, если можно решить DLP, то по известным g^a и g^b мы сможем узнать a и b и вычислить g^{ab}. Иными словами, DLP *по крайней мере* так же трудна, как CDH. Но мы не знаем наверняка, верно ли, что CDH по крайней мере так же трудна, как DLP, т. е. что обе задачи одинаково трудны. То есть DLP по отношению к CDH занимает такое же место, как задача факторизации к RSA. (Напомним, что факторизация позволила бы решить задачу RSA, но обратное, быть может, неверно.)

У протокола Диффи–Хеллмана есть еще одно сходство с RSA – DH имеет такой же уровень безопасности, как RSA, для модуля заданного размера. Например, протокол DH с 2048-битовым простым p дает примерно такую же безопасность, как RSA с 2048-битовым модулем

n, т. е. около 90 бит. Действительно, самый быстрый известный нам способ взлома CDH – решить DLP с помощью алгоритма *решета числового поля*, похожего, но не идентичного самому быстрому способу взлома RSA путем факторизации его модуля: *общему методу решета числового поля* (GNFS).

Задача Диффи–Хеллмана о распознавании

Иногда нам требуется нечто более сильное, чем предположение о трудности CDH. Например, представьте, что противник может вычислить первые 32 бита g^{ab}, зная 2048-битовые значения g^a и g^b, но не может вычислить все 2048 бит. Хотя задача CDH остается нерешенной, потому что 32 бит недостаточно для полного восстановления g^{ab}, противник все же что-то узнал о разделяемом секрете, и, возможно, это поможет ему скомпрометировать безопасность приложения.

Чтобы гарантировать, что противник не может узнать ничего о секрете g^{ab}, это значение должно быть неотличимо от случайного элемента группы. Точно так же схема шифрования считается безопасной, если шифртексты неотличимы от случайных строк. Вычислительная задача, формализующая это интуитивное представление, называется *задачей Диффи–Хеллмана о распознавании* (decisional Diffie–Hellman – DDH). Если даны g^a, g^b и значение, с вероятностью ½ равное либо g^{ab}, либо g^c для некоторого случайного c, то задача DDH заключается в том, чтобы определить, равно ли оно g^{ab} (разделяемому секрету, соответствующему g^a и g^b). Предположение о том, что противник не может эффективно решить задачу DDH, называется *предположением Диффи–Хеллмана о распознавании*.

Если DDH трудна, то CDH тоже трудна, и мы ничего не можем узнать о g^{ab}. Но если CDH можно решить, то можно решить и DDH: если дана тройка (g^a, g^b, g^c), то мы смогли бы вывести g^{ab} из g^a и g^b и проверить, равен ли результат данному g^c. Вывод: DDH фундаментально менее трудна, чем CDH, но именно трудность DDH является основным и наиболее изученным предположением в криптографии. Мы можем быть уверены, что CDH и DDH трудны при условии, что параметры протокола Диффи–Хеллмана выбраны правильно.

Другие задачи Диффи–Хеллмана

Иногда криптографы придумывают новые схемы и доказывают, что их по меньшей мере так же трудно взломать, как решить некоторые задачи, связанные с CDH или DDH, но не совпадающие с ними. В идеале мы хотели бы продемонстрировать, что вскрыть криптосистему так же трудно, как решить CDH или DDH, но при использовании передовых криптографических механизмов это не всегда возможно, прежде всего потому, что в таких схемах используются более сложные операции, чем в базовых протоколах Диффи–Хеллмана.

Например, в одной задаче, напоминающей задачу DH, дано g^a, а противник пытается вычислить $g^{1/a}$, где $1/a$ – элемент группы, обрат-

ный a (обычно группы $\mathbf{Z}_p^*$ для некоторого простого p). В другой противник пытается различить пары (g^a, g^b), зная пары (g^a, $g^{1/a}$) для случайных a и b. Наконец, в *задаче Диффи–Хеллмана о близнецах* даны g^a, g^b и g^c, а противник пытается вычислить оба значения g^{ab} и g^{ac}. Иногда такие варианты DH оказываются столь же трудными, сколь CDH или DDH, а иногда они принципиально проще – и, стало быть, дают меньшие гарантии безопасности. В качестве примера попробуйте найти связи между трудностью этих задач и трудностью CDH и DDH. (Задача Диффи–Хеллмана о близнецах на самом деле *так же* трудна, как CDH, но доказать это нелегко!)

Протоколы совместной выработки ключей

Задача Диффи–Хеллмана предназначена для построения безопасных протоколов совместной выработки разделяемого секрета двумя или более сторонами, сообщающимися по открытой сети. Этот секрет преобразуется в один или несколько *сеансовых ключей* – симметричных ключей, используемых для шифрования и аутентификации информации, которой стороны обмениваются на протяжении сеанса. Но прежде чем изучать реальные протоколы DH, нужно понимать, что делает протокол совместной выработки ключа безопасным или небезопасным и как работают более простые протоколы. Мы начнем обсуждение с широко распространенного протокола выработки ключа, не опирающегося на DH.

Пример протокола выработки ключа, не опирающегося на DH

Чтобы помочь вам составить представление о протоколе выработки ключа и его безопасности, рассмотрим протокол, используемый в стандартах связи 3G и 4G для установления связи между SIM-картой и оператором. Этот протокол часто называют *AKA* (authenticated key agreement – аутентифицированная выработка ключа). Функция Диффи–Хеллмана в нем не используется, а используются только операции с симметричным ключом. Детали немного утомительны, но в принципе протокол работает, как показано на рис. 11.1.

В этой реализации протокола на SIM-карте хранится секретный ключ K, известный оператору. Оператор начинает сеанс с выбора случайного значения R, а потом вычисляет два значения, SK и V_1, с помощью двух псевдослучайных функций, **PRF**0 и **PRF**1. Затем оператор отправляет SIM-карте сообщение, содержащее значения R и V_1, которые противник может видеть. Получив R, SIM-карта имеет все, что нужно для вычисления SK с помощью **PRF**0, что она и делает. Теперь обе стороны сеанса располагают общим ключом, SK, который противник не может определить, видя только сообщения, которыми обменивались стороны, или даже модифицируя их либо вставляя новые. SIM-карта удостоверяется, что общается именно с оператором,

повторно вычисляя V_1 с помощью **PRF**1, K и R, а затем проверяя, что вычисленное V_1 совпадает с V_1, отправленным оператором. Затем SIM-карта вычисляет проверочное значение V_2 с помощью еще одной функции, **PRF**2, которой передает K и R, и отправляет V_2 оператору. Оператор убеждается, что SIM-карта знает K, вычисляя V_2 и проверяя, что вычисленное значение совпадает с полученным.

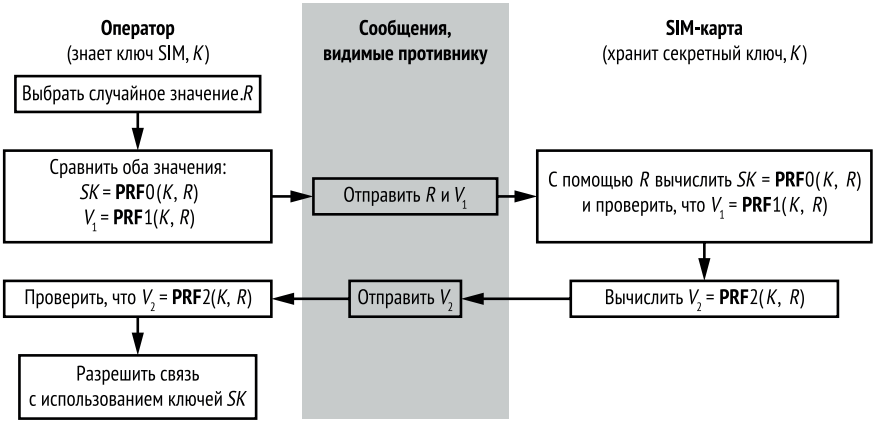

Рис. 11.1. Протокол аутентифицированной выработки ключа в стандартах связи 3G и 4G

Но этот протокол не является неуязвимым для всех типов атак: в принципе, существует способ обмануть SIM-карту с помощью атаки повторным воспроизведением. Если противник запомнил пару (R, V_1), то он сможет отправить ее SIM-карте и заставить ее поверить в то, что эта пара поступила от настоящего оператора, который знает K. Чтобы предотвратить такую атаку, протокол включает дополнительные проверки, гарантирующие, что одно и то же R не используется несколько раз.

Проблемы могут возникнуть и в случае компрометации K. Например, противник, скомпрометировавший K, может организовать атаку с человеком посередине и прослушивать все открытые сообщения. Такой противник мог бы отправлять сообщения любой стороне, притворяясь либо настоящим оператором, либо SIM-картой. Еще опаснее, что противник может записывать весь трафик, включая сообщения, которыми стороны обмениваются во время выработки ключа, а впоследствии дешифрировать этот трафик, используя запомненные значения R. Затем противник мог бы определить прошлые сеансовые ключи и использовать их для дешифрирования записанного трафика.

Модели атак на протоколы совместной выработки ключей

Не существует единого определения безопасности протоколов выработки ключей, и невозможно сказать, что данный протокол полностью

безопасен, не зная контекста и не учитывая модель атаки и цели безопасности. Например, можно было бы заявить, что рассмотренный выше протокол 3G/4G безопасен, потому что пассивный противник не сумеет определить сеансовые ключи, но можно и возразить – он не безопасен, потому что, коль скоро ключ *K* оказался в руках противника, все прошлые и будущие коммуникации скомпрометированы.

Для протоколов совместной выработки ключей есть разные аспекты безопасности, а также три основные модели атаки, зависящие от того, какая информация утекает. В порядке от самой слабой к самой сильной это *подслушивание*, *утечка данных* и *вскрытие*.

Подслушивание. Противник видит сообщения, которыми обмениваются стороны, выполняющие протокол выработки ключа, и может записывать, модифицировать, удалять и вставлять сообщения. Для защиты от подслушивания протокол не должен разглашать никакой информации о выработанном общем ключе.

Утечка данных. В этой модели противник получает сеансовый ключ и все *временные* секреты (например, *SK* в рассмотренном выше протоколе из области сотовой связи) для одного или нескольких выполнений протокола, но не секреты длительного хранения (например, *K* в том же протоколе).

Вскрытие. В этой модели противник узнает долговременный ключ одной или нескольких сторон. После вскрытия ни о какой безопасности говорить не приходится, потому что противник может притвориться одной или обеими сторонами в последующих сеансах. Тем не менее противник не должен иметь возможности восстановить секреты из сеансов, имевших место до получения ключа.

Познакомившись с моделями атак и поняв, что может сделать противник, рассмотрим цели безопасности, т. е. гарантии, которые должен давать протокол. Протокол совместной выработки ключа можно спроектировать так, что он будет удовлетворять нескольким целям безопасности. Ниже описаны четыре наиболее важные, в порядке от самой простой до самой труднодостижимой.

Аутентификация. Каждая сторона должна иметь возможность аутентифицировать другую сторону. То есть протокол должен обеспечивать *взаимную аутентификацию*. Говорить об аутентифицированной выработке ключа (AKA) можно, если протокол аутентифицирует обе стороны.

Управление ключами. Ни одна из сторон не должна иметь возможность выбрать окончательный общий секрет из заранее определенного подмножества. Рассмотренный выше протокол выработки ключа в стандарте 3G/4G не обладает этим свойством, потому что оператор выбирает значение *R*, полностью определяющее конечный разделяемый ключ.

Секретность прошлого (forward secrecy). Это свойство гарантирует, что даже если все долговременные секреты раскрыты, разде-

ляемые секреты предшествующих исполнений протокола нельзя вычислить, пусть даже противник записал все предыдущие сеансы и способен вставлять сообщения из предыдущих сеансов, возможно, в модифицированном виде. Протокол со свойством секретности прошлого гарантирует, что даже если вы обязаны предоставить свои устройства и все их секреты какому-то уполномоченному органу, то дешифрировать ранее зашифрованные сообщения все равно будет невозможно. (Протокол выработки ключа 3G/4G не обладает этим свойством.)

Стойкость к подмене при компрометации ключа (key-compromise impersonation – KCI). KCI имеет место, когда противник скомпрометировал долговременный ключ какой-то стороны и может использовать его, чтобы притвориться другой стороной. Например, в протоколе выработки ключа из стандарта 3G/4G подмена в результате компрометации ключа тривиальна, потому что обе стороны разделяют один и тот же ключ K. В идеале протокол должен предотвращать подобную атаку.

Производительность

Полезный протокол совместной выработки ключа должен быть не только безопасным, но и эффективным. При рассмотрении эффективности нужно учитывать несколько факторов, в т. ч. количество переданных сообщений, длину сообщения, вычислительные затраты на реализацию протокола и можно ли произвести какие-то вычисления заранее, чтобы сэкономить время. В общем случае более эффективен протокол, предусматривающий обмен небольшим числом коротких сообщений, и лучше, если интерактивность сведена к минимуму, т. е. ни одна сторона не должна ждать получения сообщения, прежде чем отправить следующее. Часто для измерения эффективности протокола применяется продолжительность его выполнения, выраженная в *периодах кругового обращения* (round trip), т. е. времени, необходимого для отправки сообщения и получения ответа на него.

Период кругового обращения обычно является главной причиной задержки протоколов, но объем вычислений тоже следует учитывать; чем меньше вычислений, тем лучше, и совсем хорошо, когда значительную часть вычислений можно произвести заранее.

Например, протокол совместной выработки ключа в 3G/4G предусматривает обмен двумя сообщениями по нескольку сотен битов, и они должны посылаться в определенном порядке. Для экономии времени часть вычислений можно проделать заранее, т. к. оператор может заблаговременно выбрать много значений R, вычислить соответствующие им значения SK, V_1 и V_2 и сохранить их в базе данных. В этом случае у предварительного вычисления есть то преимущество, что уменьшается шанс раскрытия долговременного ключа.

Протоколы Диффи–Хеллмана

Функция Диффи–Хеллмана – основа большинства современных протоколов выработки открытого ключа. Однако не существует единственного протокола Диффи–Хеллмана, вместо этого есть много способов использования функции DH для выработки общего секрета. В следующих разделах мы рассмотрим три таких протокола. Я буду придерживаться принятого в криптографии соглашения и называть стороны Алисой и Бобом, а противника Евой. Основание группы, фиксированное и заранее известное Алисе и Бобу, я буду обозначать g.

Анонимный протокол Диффи–Хеллмана

Анонимный протокол Диффи–Хеллмана – самый простой из таких протоколов. Он называется анонимным, потому что не требует аутентификации; у участников нет никакого идентификатора, который мог бы быть проверен другой стороной, и ни одна из сторон не хранит долговременного ключа. Алиса не может доказать Бобу, что она Алиса, и наоборот.

В анонимном протоколе Диффи–Хеллмана каждая сторона выбирает случайное значение (Алиса – a, а Боб – b) в качестве своего закрытого ключа и отправляет соответствующий открытый ключ другой стороне. На рис. 11.2 эта процедура показана более подробно.

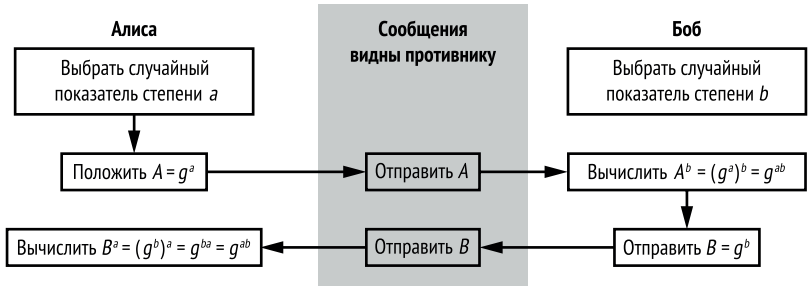

Рис. 11.2. Анонимный протокол Диффи–Хеллмана

Как видим, Алиса использует свой показатель степени a и основание группы g, чтобы вычислить $A = g^a$, и отправляет это значение Бобу. Боб получает A и вычисляет A^b, равное $(g^a)^b$. Теперь у Боба есть значение g^{ab}, и он вычисляет B по своему случайному показателю степени b и значению g. Затем он отправляет B Алисе, а она использует его для вычисления g^{ab}. Теперь у Алисы и Боба есть одно и то же значение g^{ab}, полученное выполнением похожих операций – возведением g и полученного от другой стороны значения в степень, равную своему секретному показателю. Чисто, просто, но безопасно, только когда противник совсем уж ленивый.

Анонимный алгоритм DH можно взломать с помощью атаки с человеком посередине. Противнику нужно лишь перехватить сообще-

ния и притвориться Бобом (для Алисы) и Алисой (для Боба), как показано на рис. 11.3.

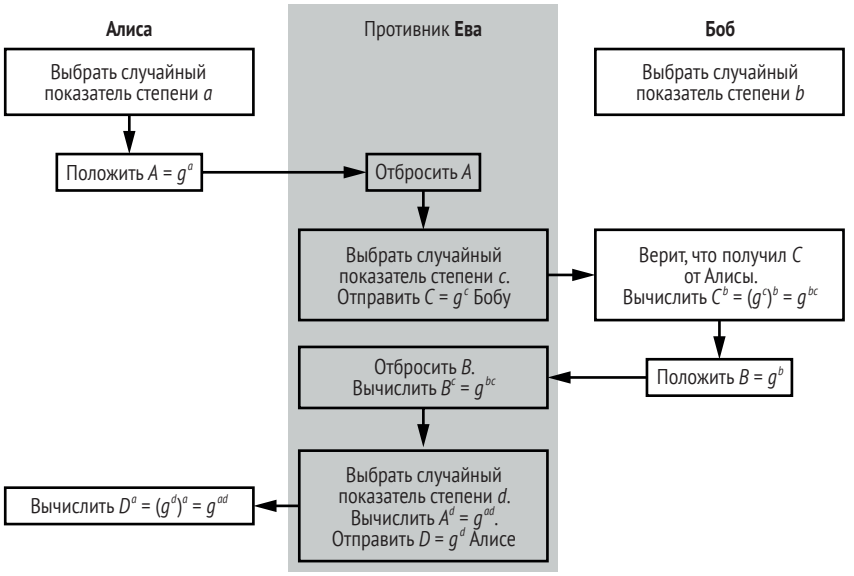

Рис. 11.3. Атака с человеком посередине на анонимный протокол Диффи–Хеллмана

Как и в предыдущем случае, Алиса и Боб выбирают случайные показатели степени, a и b. Алиса вычисляет и отправляет A, но Ева перехватывает и отбрасывает это сообщение. Ева выбирает случайный показатель степени c, вычисляет $C = g^c$ и отправляет результат Бобу. Поскольку в протоколе не предусмотрена аутентификация, Боб верит, что получил C от Алисы, и вычисляет g^{bc}. Затем Боб вычисляет B и отправляет его Алисе, но Ева снова перехватывает и отбрасывает сообщение. Теперь Ева вычисляет g^{bc}, выбирает новый показатель степени, d, вычисляет g^{ad}, вычисляет $D = g^d$ и отправляет D Алисе. Алиса тоже вычисляет g^{ad}.

В результате этой атаки противник Ева разделяет один секрет с Алисой (g^{ad}), а другой с Бобом (g^{bc}), хотя Алиса и Боб полагают, что разделяют секрет между собой. После завершения протокола Алиса будет формировать симметричные ключи по g^{ad} и шифровать ими данные, отправляемые Бобу, но Ева будет перехватывать зашифрованные сообщения, дешифрировать их, перешифровывать с помощью ключей, сформированных по g^{bc}, и отправлять Бобу – возможно, по пути изменяя открытый текст. И обо всем этом ни Алиса, ни Боб не подозревают. То есть они обречены.

Чтобы отразить эту атаку, необходим какой-то способ аутентификации сторон, чтобы Алиса могла доказать, что она настоящая Алиса, а Боб – что он настоящий Боб. К счастью, такой способ есть.

Протокол Диффи–Хеллмана с аутентификацией

Протокол Диффи–Хеллмана с аутентификацией был разработан для противостояния атаке с человеком посередине, вскрывающей анонимный протокол DH. В этом протоколе обе стороны располагают закрытым и открытым ключами, это позволяет Алисе и Бобу подписывать свои сообщения, чтобы Ева не могла отправлять сообщения от их имени. Подписи вычисляются не функцией DH, а по схеме подписания открытым ключом, например RSA-PSS. В результате, чтобы успешно подписывать сообщения от имени Алисы, Ева должна была бы подделать ее подпись, что невозможно, если схема подписания безопасна.

На рис. 11.4 показано, как работает протокол DH с аутентификацией.

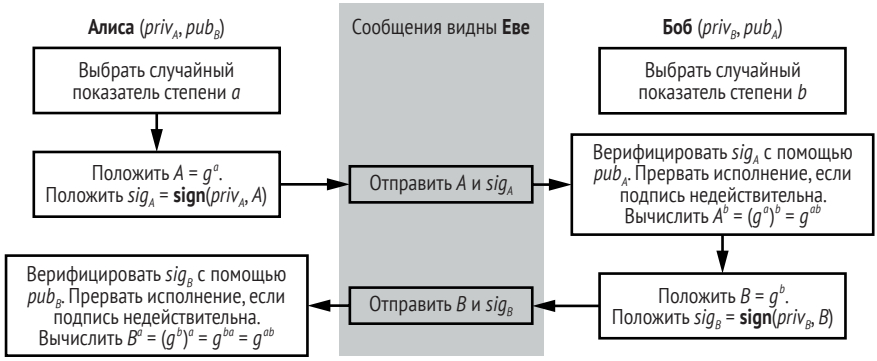

Рис. 11.4. Протокол Диффи–Хеллмана с аутентификацией

Надпись **Alice** ($priv_A$, pub_B) в первой строке означает, что Алиса хранит свой закрытый ключ, *privA*, и открытый ключ Боба, pub_B. Эта пара ключей *priv/pub* называется *долговременным ключом*, потому что фиксирована заранее и остается неизменной при всех выполнениях протокола. Разумеется, долговременные ключи следует хранить в секрете, тогда как открытые ключи считаются известными противнику.

Сначала Алиса и Боб выбирают случайные показатели степени, a и b, как в анонимном протоколе DH. Затем Алиса вычисляет A и подпись sig_A, основываясь на сочетании своей функции подписания **sign**, своего закрытого ключа $priv_A$ и A. Далее Алиса отправляет A и sig_A Бобу, который верифицирует sig_A ее открытым ключом pub_A. Если подпись недействительна, то Боб понимает, что сообщение пришло не от Алисы, и отбрасывает A.

Если подпись правильна, то Боб вычисляет g^{ab} по A и своему случайному показателю степени b. Затем он вычисляет B и свою подпись, основываясь на сочетании функции подписания **sign**, своего закрытого ключа $priv_B$ и B. Далее Боб отправляет B и sig_B Алисе, которая пытается верифицировать sig_B открытым ключом Боба pub_B. Алиса вычисляет g^{ab}, только если подпись Боба действительна.

Защита от подслушивания

Протокол DH с аутентификацией защищен от подслушивания, потому что противник не может добыть ни единого бита информации о разделяемом секрете g^{ab}, не зная показателей степени. DH с аутентификацией обеспечивает также секретность прошлого: даже если противник скомпрометирует в какой-то момент любую сторону, как в модели атаки со *вскрытием*, он узнает закрытые ключи подписания, но не будет знать ни один из эфемерных показателей степени DH, а значит, не сможет узнать ни один из прошлых разделяемых секретов.

DH с аутентификацией также не дает ни одной из сторон управлять значениями разделяемого секрета. Алиса не может изготовить специальное значение a, чтобы предсказать значение g^{ab}, потому что не контролирует b, влияющее на g^{ab} в той же мере, что и a. (Есть одно исключение: если бы Алиса выбрала $a = 0$, то g^{ab} было бы равно 1 при любом b. Но 0 не является допустимым значением, и протокол должен его отвергать.)

Тем не менее DH с аутентификацией не является безопасным относительно всех типов атак. Действительно, Ева может запоминать предыдущие значения A и sig_A и впоследствии воспроизводить их Бобу, чтобы притвориться Алисой. Тогда Боб поверит, что разделяет секрет с Алисой, хотя на самом деле это не так, – и это несмотря на то, что Ева не смогла узнать секрет. На практике этот риск можно устранить, добавив процедуру *подтверждения ключа*, в ходе которой Алиса и Боб доказывают друг другу, что владеют общим секретом. Например, Алиса и Боб могут подтвердить ключ, отправив соответственно **Hash**($pub_A || pub_B, g^{ab}$) и **Hash**($pub_B || pub_A, g^{ab}$), где **Hash** – некоторая хеш-функция. Когда Боб получит **Hash**($pub_A || pub_B, g^{ab}$), а Алиса – **Hash**($pub_B || pub_A, g^{ab}$), обе стороны смогут проверить правильность хеш-значений, используя pub_A, pub_B и g_{ab}. Благодаря разному порядку конкатенации ключей ($pub_A || pub_B$ и $pub_B || pub_A$) гарантируется, что Алиса и Боб отправят разные значения и, значит, противник не сможет притвориться Алисой, скопировав хеш-значение Боба.

Защита от утечки данных

Уязвимость DH с аутентификацией к утечке данных – более серьезная проблема. В атаке этого типа противник узнает значение эфемерных секретов с коротким временем жизни (а именно показателей степени a и b) и использует эту информацию для подмены одной из сторон. Если Ева сможет узнать значение показателя степени a вместе с соответствующими значениями A и sig_A, отправленными Бобу, то она сможет инициировать новое выполнение протокола и притвориться Алисой, как показано на рис. 11.5.

В этом сценарии Ева узнает значение a и воспроизводит соответствующее A и подпись sig_A, притворяясь Алисой. Боб верифицирует подпись, вычисляет g^{ab} по A и отправляет B и sig_B, которые Ева использует для вычисления g^{ab}, пользуясь все тем же украденным a.

В результате оба владеют разделяемым секретом. Боб теперь думает, что общается с Алисой.

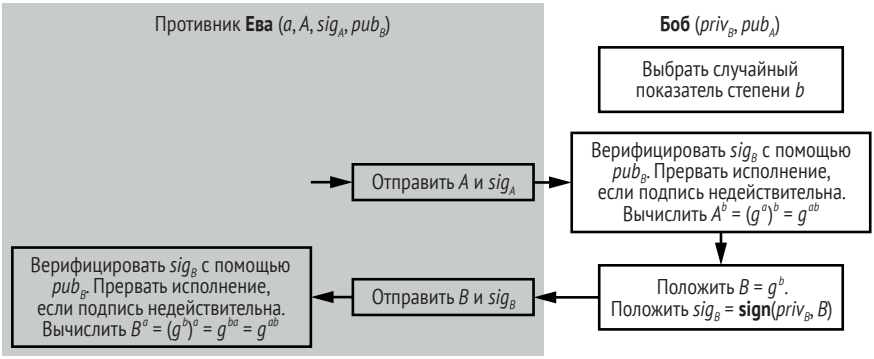

Рис. 11.5. Атака с подменой на протокол Диффи–Хеллмана с аутентификацией

Один из способов защитить DH с аутентификацией от утечки эфемерных секретов – включить долговременные секреты в вычисление разделяемого секрета, так чтобы разделяемый секрет нельзя было определить, не зная долговременного.

Протокол Менезеса–Кью–Вэнстоуна

Протокол Менезеса–Кью–Вэнстоуна (*MQV*) – веха в истории протоколов на основе DH. Спроектированный в 1998 году, MQV одобрен для защиты самых важных активов, АНБ включило его в свой Комплект B – набор алгоритмов, предназначенных для защиты секретной информации. (В конечном итоге АНБ исключило MQV предположительно потому, что он не использовался. Ниже я еще поговорю о причинах.)

MQV – это усовершенствованный протокол Диффи–Хеллмана. Он безопаснее DH с аутентификацией и лучше последнего по производительности. В частности, MQV разрешает пользователям отправлять только два сообщения, независимо друг от друга и в произвольном порядке. Есть и другие преимущества: пользователи могут обмениваться более короткими сообщениями, чем в DH с аутентификацией, и не обязаны отправлять явно сообщения с подписью и о ее верификации. Иными словами, необязательно использовать схему подписания в дополнение к функции Диффи–Хеллмана.

Как и в DH с аутентификацией, в MQV Алиса и Боб хранят свой долговременный закрытый ключ, а также долговременный открытый ключ другой стороны. Разница в том, что в MQV ключи используются не для подписания, а состоят из закрытого показателя степени x и открытого значения g^x. На рис. 11.6 показано, как работает протокол MQV.

На рис. 11.6 x и y – долговременные закрытые ключи Алисы и Боба соответственно, а X и Y – их открытые ключи. Вначале Боб и Алиса располагают своими собственными закрытыми ключами и открыты-

ми ключами друг друга, равными g в степени, равной закрытому ключу. Каждая сторона выбирает случайный показатель степени, после чего Алиса вычисляет A и отправляет Бобу, а Боб вычисляет B и отправляет Алисе. Получив эфемерный открытый ключ Боба B, Алиса объединяет его со своим долговременным закрытым ключом x, своим эфемерным закрытым ключом a и долговременным открытым ключом Боба Y, вычисляя выражение $(B \times Y^B)^{a+xA}$, как показано на рис. 11.6. Преобразование этого выражения дает

$$(B \times Y^B)^{a+xA} = (g^b \times (g^y)^B)^{a+xA} = (g^{b+yB})^{a+xA} = g^{(b+yB)(a+xA)}.$$

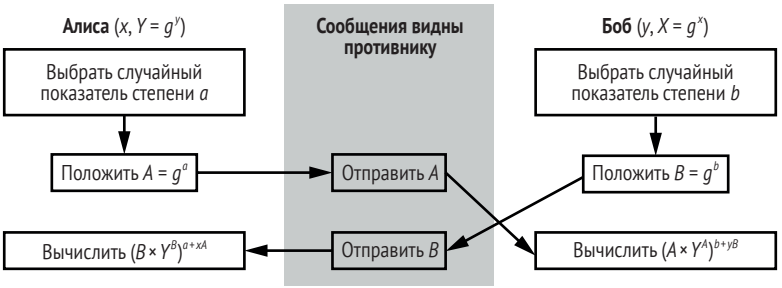

Рис. 11.6. Протокол MQV

На рис. 11.6 x и y – долговременные закрытые ключи Алисы и Боба Тем временем Боб вычисляет выражение $(A \times X^A)^{b+yB}$, и легко проверить, что оно равно выражению, вычисленному Алисой:

$$(A \times X^A)^{b+yB} = (g^a \times (g^x)^A)^{b+yB} = (g^{a+xA})^{b+yB} = g^{(a+xA)(b+yB)} = g^{(b+yB)(a+xA)}.$$

Как видим, Алиса и Боб получают одинаковое значение, $g^{(b+yB)(a+xA)}$, т. е. разделяют общий секрет.

В отличие от DH с аутентификацией, протокол MQV невозможно взломать из-за простой утечки эфемерных секретов. Знание a или b не позволит противнику добыть окончательный разделяемый секрет, потому что для его вычисления нужно знать долговременные закрытые ключи.

Что произойдет при самой сильной модели атаки, со вскрытием, когда скомпрометирован долговременный ключ? Если Ева скомпрометирует долговременный закрытый ключ Алисы, x, то ранее выработанные разделяемые секреты останутся в безопасности, потому что в их вычислении участвовали также эфемерные закрытые ключи Алисы.

Однако MQV не дает *совершенной* секретности прошлого из-за следующей атаки. Предположим, что Ева перехватила сообщение Алисы A и заменила его своим сообщением $A = g^a$ с некоторым a, выбранным Евой. Тем временем Боб отправляет B Алисе (и Ева запоминает значение B) и вычисляет разделяемый ключ. Если впоследствии Ева скомпрометирует долговременный закрытый ключ Алисы x, то сможет определить, какой ключ вычислил Боб в этом сеансе. Это на-

рушает секретность прошлого, потому что Ева смогла восстановить разделяемый ключ для предыдущего исполнения протокола. Однако на практике этот риск можно устранить, добавив шаг подтверждения ключа, благодаря которому Алиса и Боб могут осознать, что разделяют не один и тот же ключ. После этого они прервут выполнение протокола, не приступая к выводу сеансовых ключей.

Несмотря на элегантность и безопасность, MQV редко используется на практике. Связано это, в частности, с тем, что раньше он был обременен патентами, мешавшими широкому распространению. Другая причина – тот факт, что правильно реализовать MQV труднее, чем кажется. В общем, с учетом повышенной сложности преимущества MQV с точки зрения безопасности не кажутся такими весомыми, чтобы отказаться от более простого протокола DH с аутентификацией.

Какие возможны проблемы

Протоколы Диффи–Хеллмана подвержены впечатляющим провалам – и не одним способом. В следующих разделах я расскажу о самых распространенных.

Пренебрежение хешированием разделяемого секрета

Я упоминал, что разделяемый секрет, который вырабатывается в результате обмена сообщениями в сеансе DH (в наших примерах – g^{ab}), служит исходной точкой для вывода сеансовых ключей, но сам ключом не является. И не должен являться. Симметричный ключ должен казаться случайным, все его биты должны принимать значение 0 или 1 с одинаковой вероятностью. Но g^{ab} – вовсе не случайная строка; это случайный элемент некоторой математической группы, и его биты могут иметь статистическое смещение. Случайный элемент группы отличается от случайной строки битов.

Представьте, к примеру, что мы работаем с мультипликативной группой $\mathbf{Z}^*_{13} = \{1, 2, 3, \ldots, 12\}$, для которой $g = 2$ является порождающим элементом, т. е. множество значений g^i, $i = 1, 2, \ldots, 12$, совпадает со всем множеством элементов $\mathbf{Z}^*_{13}$: $g^1 = 2$, $g^2 = 4$, $g^3 = 8$, $g^4 = 3$ и т. д. При случайном выборе показателя степени g мы получим случайный элемент $\mathbf{Z}^*_{13}$, но представление элемента $\mathbf{Z}^*_{13}$ в виде 4-битовой строки не будет иметь равномерного распределения: не все биты принимают значение 0 или 1 с одинаковой вероятностью. В $\mathbf{Z}^*_{13}$ у семи элементов (с номерами от 1 до 7) старшим битом является 0, и только у пяти (от 8 до 12) он равен 1. Следовательно, этот бит равен 0 с вероятностью $7/12 \approx 0.58$, тогда как в идеале вероятность должна быть равна 0.5. Более того, 4-битовые последовательности 1101, 1110 и 1111 вообще ни разу не встречаются.

Чтобы избежать такого смещения сеансовых ключей, сформированных по разделяемому секрету DH, следует использовать криптографическую функцию хеширования, например BLAKE2 или SHA-3,

а еще лучше функцию формирования ключа (KDF). Примером построения KDF может служить HKDF, или KDF на основе HMAC (описанная в RFC 5869), но в настоящее время у BLAKE2 и SHA-3 есть режимы, в которых они ведут себя как KDF.

Унаследованный протокол Диффи–Хеллмана в TLS

Протокол TLS отвечает за безопасность HTTPS-сайтов, а равно за безопасность почтового протокола SMTP. TLS принимает несколько параметров, в т. ч. тип используемого протокола Диффи–Хеллмана, и большинство реализаций TLS ради обратной совместимости до сих пор поддерживают анонимный протокол DH, несмотря на его небезопасность.

Небезопасные параметры группы

В январе 2016 года лица, отвечающие за сопровождение пакета OpenSSL, исправили весьма серьезную уязвимость (CVE-2016-0701), которая позволяла противнику эксплуатировать небезопасные параметры протокола Диффи–Хеллмана. Главной причиной этой уязвимости был тот факт, что OpenSSL позволял работать с небезопасными параметрами группы (а именно с небезопасным простым p), вместо того чтобы возбудить исключение и прервать выполнение протокола еще до каких-либо арифметических операций.

Суть дела в том, что OpenSSL принимал простое число p, для которого мультипликативная группа $\mathbf{Z}_p^*$ (в которой и производятся все операции DH) содержала подгруппы малого размера. В начале этой главы мы говорили, что существование малых подгрупп в криптографическом протоколе – зло, потому что сильно ограничивает множество возможных значений разделяемых секретов. Хуже того, противник может изготовить показатель степени x, который в сочетании с открытым ключом жертвы g^y будет раскрывать информацию о закрытом ключе y и в конце концов раскроет его целиком.

Хотя открытие уязвимости датируется 2016 годом, принцип атаки восходит еще к статье Lim and Lee «A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup» 1997 года. Исправить эту ошибку просто: прежде чем принять простое число p в качестве модуля группы, протокол должен проверить, что p безопасно, т. е. что число $(p - 1)/2$ тоже простое, тогда у группы $\mathbf{Z}_p^*$ не будет малых подгрупп и атака потерпит неудачу.

Для дополнительного чтения

Отмечу несколько вещей, которым не нашлось места в этой главе, но о которых полезно почитать.

Более глубоко изучить протоколы совместной выработки ключей Диффи–Хеллмана позволит чтение стандартов и официальных доку-

ментов, включая ANSI X9.42, RFC 2631 и RFC 5114, IEEE 1363 и NIST SP 800-56A. Они обеспечивают интероперабельность и содержат рекомендации о параметрах группы.

Если вы хотите узнать больше о передовых протоколах DH (в частности, MQV и родственных ему HMQV и OAKE) и их аспектах безопасности (например, атаках на разделение неизвестного ключа и атаках на представление группы), прочитайте статью Hugo Krawczyk «HMQV: A High-Performance Secure Diffie–Hellman Protocol» 2005 года (*https://eprint.iacr.org/2005/176/*) и статью Andrew C. Yao and Yunlei Zhao «A New Family of Implicitly Authenticated Diffie–Hellman Protocols» 2011 года (*https://eprint.iacr.org/2011/035/*). Обратите внимание, что в этих статьях операции Диффи–Хеллмана записываются не так, как в этой главе. Например, разделяемый секрет представлен в виде xP, а не g^x. Вообще, умножение заменено сложением, а возведение в степень – умножением. Причина в том, что эти протоколы обычно определяются не над группами целых чисел, а над *эллиптическими кривыми*, о которых пойдет речь в главе 12.

12

ЭЛЛИПТИЧЕСКИЕ КРИВЫЕ

Появление *криптографии на эллиптических кривых* (elliptic curve cryptography – ECC), или *эллиптической криптографии*, в 1985 году произвело революцию в привычной уже криптографии с открытым ключом. ECC – более мощная и эффективная альтернатива таким схемам, как RSA и классический протокол Диффи–Хеллмана (стойкость ECC с 256-битовым ключом выше, чем у RSA с 4096-битовым ключом), но она и сложнее.

Как и RSA, ECC подразумевает умножение больших чисел, но, в отличие от RSA, это делается для того, чтобы скомбинировать точки на некоторой математической кривой, называемой *эллиптической* (кстати, никакого отношения к эллипсу она не имеет). Если это кажется вам недостаточно сложным, добавлю, что существует много типов эллиптических кривых – простых и сложных, эффективных и неэффективных, безопасных и небезопасных.

Хотя ECC была открыта в 1985 году, органы стандартизации не проявляли к ней интереса до начала 2000-х годов, а в основные пакеты программы она была включена еще позже: в OpenSSL в 2005 году, а OpenSSH дожидался своей очереди аж до 2011 года. Но в современных системах есть мало причин не использовать ECC, и она действительно используется в технологии биткойна и во многих компонентах безопасности устройств Apple. Эллиптические кривые позволяют выполнять типичные операции криптографии с открытым ключом – шифрование, подписание и совместную выработку ключа – быстрее классических схем. Большинство криптографических приложений, опирающихся на задачу о дискретном логарифме (DLP), работают и с эллиптическим аналогом (ECDLP) – за одним заметным исключением: протокол Secure Remote Password (SRP).

В этой главе мы будем говорить о приложениях ECC и о том, почему лучше использовать ECC, чем RSA или классический протокол Диффи–Хеллмана. А заодно о том, как правильно выбрать эллиптическую кривую для своего приложения.

Что такое эллиптическая кривая?

Эллиптическая кривая – это *кривая* на плоскости, т. е. группа точек с координатами x и y. Уравнение кривой определяет множество всех принадлежащих ей точек. Например, кривая $y = 3$ – горизонтальная прямая, состоящая из всех точек с ординатой 3. Кривые вида $y = ax + b$ с фиксированными a и b – это прямые линии; $x^2 + y^2 = 1$ – окружность радиуса 1 с центром в начале координат и т. д.

Эллиптические кривые, используемые в криптографии, обычно описываются уравнением вида $y^2 = x^3 + ax + b$ (так называемая *форма Вейерштрасса*), где постоянные a и b определяют форму кривой. Например, на рис. 12.1 показана эллиптическая кривая, описываемая уравнением $y^2 = x^3 - 4x$.

Примечание *В этой главе нас будет интересовать самый простой и распространенный тип эллиптических кривых, а именно описываемые уравнением вида $y^2 = x^3 + ax + b$, однако существуют эллиптические кривые, описываемые другими уравнениями. Например, кривые Эдвардса описываются уравнением вида $x^2 + y^2 = 1 + dx^2y^2$. Кривые Эдвардса иногда используются в криптографии (например, в схеме Ed25519).*

На рис. 12.1 показаны все точки кривой на отрезке от –3 до 4 – как на левой ветви, напоминающей окружность, так и на правой, похожей на параболу. Это точки, координаты (x, y) которых удовлетворяют уравнению $y^2 = x^3 - 4x$. Например, при $x = 0$ $y^2 = x^3 - 4x = 0^3 - 4 \times 0 = 0$, поэтому $y = 0$ является решением уравнения и, значит, точка (0, 0) принадлежит кривой. Аналогично при $x = 2$ решением будет $y = 0$, т. е. точка (2, 0) тоже принадлежит кривой.

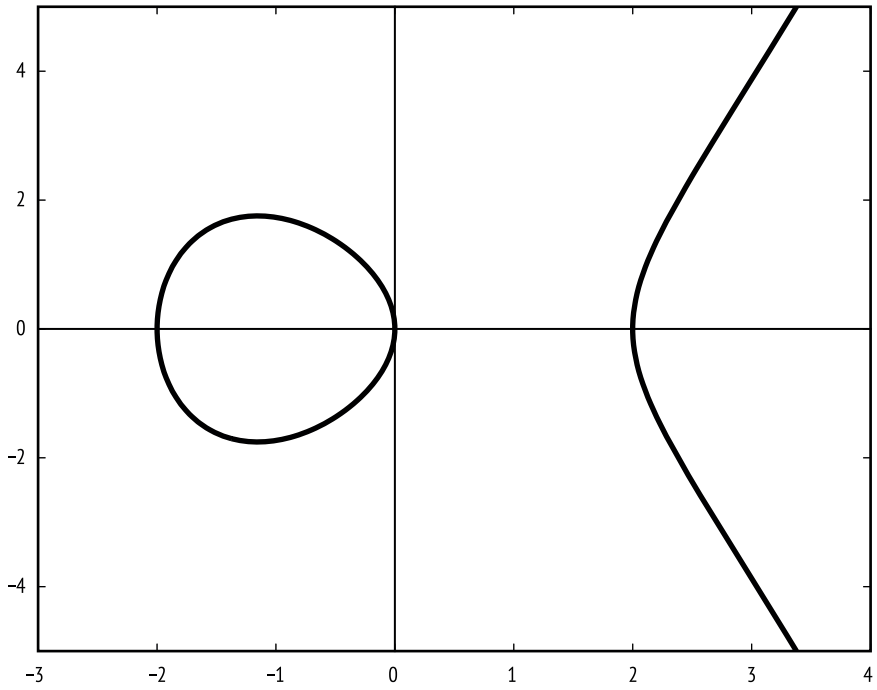

Рис. 12.1. Эллиптическая кривая над множеством вещественных чисел, описываемая уравнением $y^2 = x^3 - 4x$

Важно отличать точки, принадлежащие кривой, от всех остальных точек, потому что в криптографии мы работаем только с точками на эллиптической кривой, а остальные часто представляют угрозу для безопасности. Заметим, однако, что уравнение кривой не всегда имеет решения, по крайней мере на числовой плоскости. Например, чтобы найти точки с абсциссой $x = 1$, мы должны решить уравнение $y^2 = x^3 - 4x$, которое для $x = 1$ принимает вид $y^2 = -3$. Но это уравнение не имеет решения в вещественных числах. (На комплексной плоскости решение существует, но эллиптическая криптография имеет дело только с натуральными числами, точнее с целыми числами по модулю простого числа.) Отсутствие решения для $x = 1$ означает, что на кривой нет точек с такой абсциссой, что наглядно видно на рис. 12.1.

А если попробовать значение $x = -1$? В этом случае уравнение принимает вид $y^2 = -1 + 4 = 3$ и имеет два решения: $y = \sqrt{3}$ и $y = -\sqrt{3}$. Возведение числа в квадрат всегда дает положительный результат, поэтому $y^2 = (-y)^2$ для любого вещественного y, и, как легко видеть, кривая на рис. 12.1 симметрична относительно оси x (как и все эллиптические кривые вида $y^2 = x^3 + ax + b$).

Эллиптические кривые над множеством целых чисел

А теперь неожиданный поворот: кривые, применяемые в эллиптической криптографии, выглядят совсем не так, как показано на рис. 12.1,

а как показано на рис. 12.2 – в виде облака точек, а не кривой. Что же здесь происходит?

Рисунки 12.1 и 12.2 на самом деле основаны на одном и том же уравнении кривой, $y^2 = x^3 - 4x$, но показывают ее точки относительно разных множеств чисел. На рис. 12.1 мы видим точки кривой над множеством *вещественных чисел*, включающим отрицательные числа, дроби и т. д. Поскольку это непрерывная кривая, то показаны точки при $x = 2.0$, $x = 2.1$, $x = 2.00002$ и т. д. С другой стороны, на рис. 12.2 показаны только целые числа, удовлетворяющие уравнению кривой, и никаких дробей. Точнее, на рис. 12.2 показана кривая $y^2 = x^3 - 4x$ над множеством целых чисел по *модулю 191*: 0, 1, 2, 3, ..., 190. Это множество чисел обозначается $\mathbf{Z}_{191}$. (В числе 191 нет ничего особенного, кроме одного – оно простое. Я выбрал небольшое число, чтобы на графике было не слишком много точек.) Таким образом, обе координаты точек на рис. 12.2 – целые числа, не превосходящие 191 и удовлетворяющие уравнению $y^2 = x^3 - 4x$. Например, для $x = 2$ имеем $y^2 = 0$, поэтому $y = 0$ – решение и, значит, точка (2, 0) лежит на кривой.

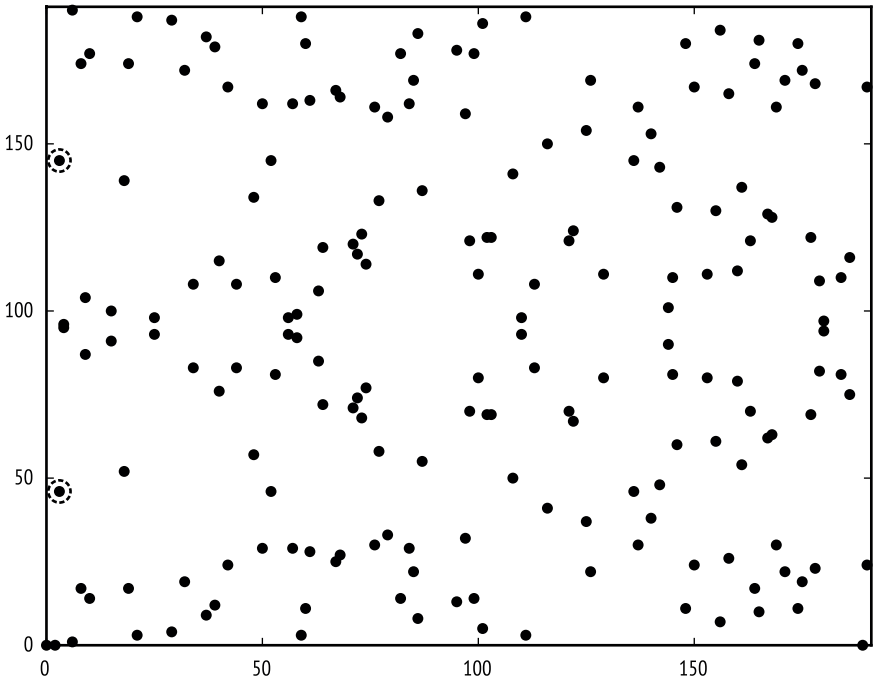

Рис. 12.2. Эллиптическая кривая с уравнением $y^2 = x^3 - 4x$ над множеством $\mathbf{Z}_{191}$ целых чисел по модулю 191

А если $x = 3$? Тогда мы получаем уравнение $y^2 = 27 - 12 = 15$, которое допускает два решения в $\mathbf{Z}_{191}$: 46 и 145, поскольку $46^2 \bmod 191 = 15$ и $145^2 \bmod 191 = 1$. Следовательно, точки (3, 46) и (3, 145) принадлежат кривой (обе они обведены кружочком в левой части рис. 12.2).

Примечание *На рис. 12.2 рассмотрены точки из множества* $\mathbf{Z}_{191} = \{0, 1, 2, \ldots, 190\}$*, включающего нуль. Этим оно отличается от групп, обозначаемых* $\mathbf{Z}_p^*$ *(обратите внимание на верхнюю звездочку), которые мы обсуждали в контексте RSA и протокола Диффи–Хеллмана. Причина этого отличия в том, что здесь мы не только умножаем, но и складываем числа, поэтому хотим, чтобы множество включало единичный элемент относительно сложения (а именно 0, т. к.* $x + 0 = x$ *для любого* $x \in \mathbf{Z}_{191}$*). Кроме того, для каждого числа x имеется обратное относительно сложения, обозначаемое* $-x$ *и такое, что* $x + (-x) = 0$*. Например, обратным к* 100 *в множестве* $\mathbf{Z}_{191}$ *является* 91*, потому что* $100 + 91 \bmod 191 = 0$*. Такое множество чисел, в котором определены операции сложения и умножения и для каждого элемента x существует обратный относительно сложения (обозначаемый* $-x$*), а также обратный относительно умножения (обозначаемый* $1/x$*), называется полем. Если поле содержит конечное число элементов, как* $\mathbf{Z}_{191}$ *и вообще все поля, используемые в эллиптической криптографии, то оно называется конечным полем.*

Сложение и умножение точек

Мы видели, что координаты (x, y) точек на эллиптической кривой удовлетворяют уравнению $y^2 = x^3 + ax + b$. В этом разделе мы рассмотрим, как складывать точки на эллиптической кривой по *правилу сложения*.

Сложение двух точек

Пусть требуется сложить две точки на эллиптической кривой, P и Q, с целью получить новую точку R, являющуюся их суммой. Проще всего для этого воспользоваться геометрическим правилом: провести прямую, соединяющую P и Q, и найти на кривой еще одну точку пересечения с этой прямой. Тогда суммой P и Q будет точка, симметричная этой точке пересечения относительно оси x. Например, на рис. 12.3 прямая, соединяющая P и Q, пересекает кривую в точке, расположенной между P и Q, и точка $P + Q$ имеет такую же абсциссу, как эта третья точка, но противоположную ординату.

Это геометрическое правило простое, но не дает численных значений координат точки R. Ее координаты (x_R, y_R) вычисляются по координатам (x_P, y_P) точки P и (x_Q, y_Q) точки Q по формулам $x_R = m^2 - x_P - x_Q$, $y_R = m(x_P - x_R) - y_P$, где $m = (y_Q - y_P)/(x_Q - x_P)$ – угловой коэффициент прямой, соединяющей P и Q.

К сожалению, эти формулы и прием с проведением прямой, как на рис. 12.3, работают не всегда. Например, если $P = Q$, то нельзя провести прямую через две точки (точка-то всего одна), а если $P = -P$, то эта прямая не пересечет кривую во второй раз, так что нечего будет отражать относительно оси x. Мы рассмотрим эти случаи в следующем разделе.

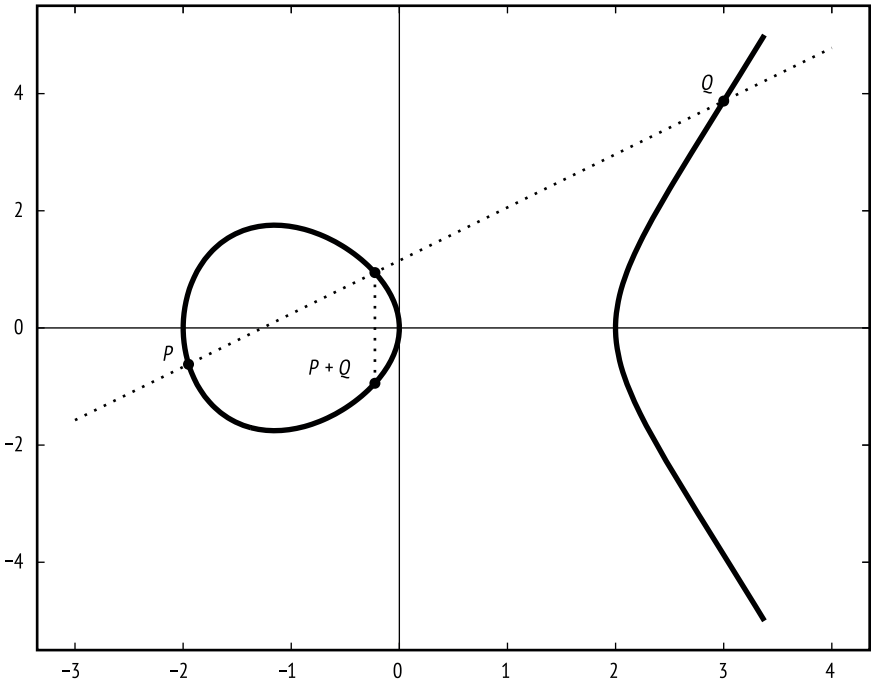

Рис. 12.3. Общий случай применения геометрического правила сложения точек на эллиптической кривой

Сложение двух противоположных точек

Противоположной к точке $P = (x_P, y_P)$ называется точка $-P = (x_P, -y_P)$, являющаяся отражением P относительно оси x. Для любой точки P мы говорим, что $P + (-P) = O$, где O называется *бесконечно удаленной точкой*. Как видно на рис. 12.4, прямая, проходящая через P и $-P$, уходит в бесконечность, не пересекая кривой. (Бесконечно удаленная точка – это воображаемая точка, принадлежащая любой эллиптической кривой; для эллиптических кривых это то же, что нуль для целых чисел.)

Удвоение точки

Если $P = Q$ (т. е. P и Q занимают одно и то же место), то сложение P и Q эквивалентно вычислению суммы $P + P$, которая обозначается $2P$. Поэтому такая операция сложения называется *удвоением*.

Однако для нахождения координат результата $R = 2P$ мы не можем воспользоваться геометрическим правилом из предыдущего раздела, потому что невозможно провести прямую, соединяющую точку с самой собой. Вместо этого мы проводим прямую, *касательную* к кривой в точке P, тогда $2P$ будет противоположна той точке, в которой эта прямая пересекается с кривой (см. рис. 12.5).

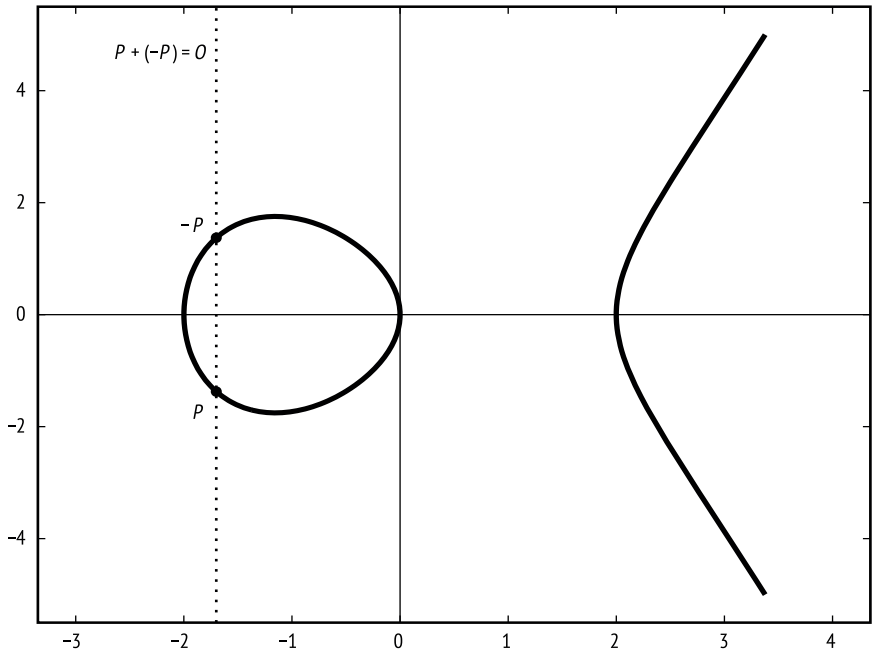

Рис. 12.4. Геометрическое правило сложения точек на эллиптической кривой, дополненное операцией P + (–P) = O в случае, когда прямая, проходящая через две точки, не пересекает кривую

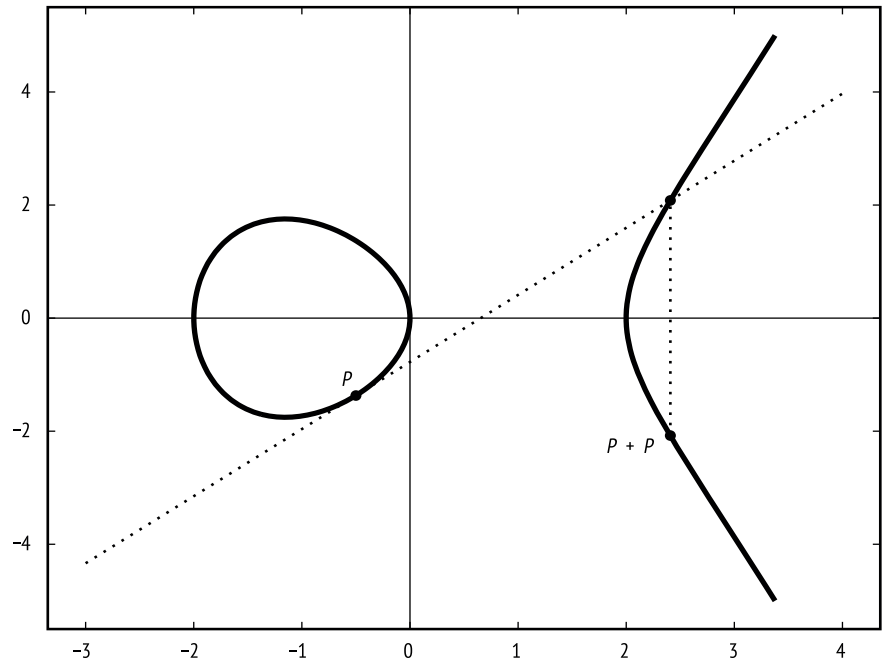

Рис. 12.5. Геометрическое правило сложения точек на эллиптической кривой, иллюстрирующее операцию удвоения P + P

Формула для определения координат (x_R, y_R) точки $R = P + P$ немного отличается от формулы в случае разных P и Q. Базовые формулы $x_R = m^2 - x_P - x_Q$, $y_R = m(x_P - x_R) - y_P$ остаются такими же, но значение m другое; оно равно $(3x_P^2 + a) / 2y_P$, где a – параметр кривой – коэффициент при x в уравнении $y^2 = x3 + ax + b$.

Умножение

Чтобы умножить точку на эллиптической кривой на целое число k, нужно найти точку kP, сложив P с собой $k - 1$ раз. То есть $2P = P + P$, $3P = P + P + P$ и т. д. Для нахождения координат x и y точки kP следует применить описанное выше правило сложения несколько раз.

Однако для эффективного вычисления kP наивного применения правила сложения $k - 1$ раз отнюдь не достаточно. Например, если k велико (скажем, порядка 2^{256}), как то бывает в криптографических схемах на основе эллиптических кривых, то выполнить $k - 1$ сложений попросту невозможно.

Но есть один трюк: экспоненциального ускорения можно достичь, применив ту же технику, что была описана в разделе «Алгоритм быстрого возведения в степень» главы 10 для вычисления $x^e \bmod n$. Например, чтобы вычислить $8P$ за три сложения вместо семи при наивном способе, нужно сначала вычислить $P2 = P + P$, затем $P_4 = P_2 + P_2$ и, наконец, $P_4 + P_4 = 8P$.

Группы эллиптических кривых

Поскольку точки можно складывать, множество точек на эллиптической кривой образует группу[1]. Согласно определению группы (см. раздел «Что такое группа?» главы 9), если точки P и Q принадлежат данной кривой, то и точка $P + Q$ тоже ей принадлежит.

Далее, поскольку сложение ассоциативно, имеем $(P + Q) + R = P + (Q + R)$ для любых точек P, Q и R. В группе эллиптической кривой единичный элемент называется бесконечно удаленной точкой и обозначается O; это такой элемент, что $P + O = P$ для любой точки P. Для каждой точки $P = (x_P, y_P)$ имеется обратная, $-P = (x_P, -y_P)$, такая, что $P + (-P) = O$.

На практике большинство криптосистем на основе эллиптических кривых работают с координатами x и y, которые являются числами по простому модулю p (т. е. элементами конечного поля $\mathbf{Z}_p$). Как безопасность RSA зависит от размера используемых чисел, так и безопасность эллиптической криптосистемы зависит от количества точек на кривой. Но откуда мы знаем количество точек на эллиптической кривой, или ее *мощность*? Это зависит от кривой и значения p.

---

[1] Аргументация автора весьма странна. Ему бы надо доказать, что для множества точек на эллиптической кривой выполняются аксиомы группы, а он вместо этого постулирует, что это группа, и отсюда выводит замкнутость операции сложения, ее ассоциативность и т. д. – *Прим. перев.*

Согласно эвристическому правилу, на кривой имеется приблизительно p точек, но можно вычислить и точное значение, воспользовавшись алгоритмом Шуфа. Этот алгоритм реализован в программе SageMath. Так, в листинге 12.1 показано, как с его помощью подсчитать количество точек на кривой $y^2 = x^3 - 4x$ над полем $\mathbf{Z}_{191}$, показанной на рис. 12.1.

Листинг 12.1. Вычисление мощности кривой (количества точек на ней)

```
sage: Z = Zmod(191)
sage: E = EllipticCurve(Z, (-4,0))
sage: E.cardinality()
192
```

Здесь мы сначала определяем переменную `Z` как множество целых чисел по модулю 191, а затем переменную `E` как эллиптическую кривую над `Z` с коэффициентами –4 и 0. Наконец, мы вычисляем количество точек на этой кривой, которое называется *мощностью*, *порядком группы* или просто *порядком*. Заметим, что мощность включает и бесконечно удаленную точку O.

Задача ECDLP

В главе 9 мы познакомились с задачей о дискретном логарифме (DLP): найти для данного основания g и большого простого числа p такое число y, что $x = g^y \bmod p$. В эллиптической криптографии есть похожая задача: для данных точек P и Q найти такое число k, что $Q = kP$. Это так называемая задача дискретного логарифмирования на эллиптической кривой (*elliptic curve discrete logarithm problem – ECDLP*). (Вместо чисел в задачах на эллиптических кривых фигурируют точки, а вместо возведения в степень используется умножение.)

Вся эллиптическая криптография построена вокруг задачи ECDLP, которая, как и DLP, считается трудной и не поддается криптоанализу с момента постановки в 1985 году. Между ECDLP и классической DLP есть важное различие: ECDLP позволяет работать с меньшими числами без снижения уровня безопасности.

В общем случае, если длина p составляет n бит, то уровень безопасности равен примерно $n/2$ бит. Например, эллиптическая кривая над полем $\mathbf{Z}_p$ с 256-битовым p обеспечивает уровень безопасности около 128 бит. Для сравнения: чтобы достичь такого же уровня при использовании DLP или RSA, понадобились бы числа, состоящие из тысяч бит. Использование меньших чисел – одна из причин, почему ECC зачастую работает быстрее, чем RSA или классический алгоритм Диффи–Хеллмана.

Один из способов решить ECDLP – найти коллизию между точками $c_1P + d_1Q$ и $c_2P + d_2Q$. Здесь точки P и Q таковы, что $Q = kP$ для некоторого неизвестного k, а c_1, d_1, c_2 и d_2 – числа, необходимые для нахождения k.

Как и для хеш-функций – предмета обсуждения в главе 6, – коллизия имеет место, когда два разных входа дают одинаковый выход. Поэтому для решения ECDLP мы должны найти точки, для которых справедливо равенство:

$$c_1P + d_1Q = c_2P + d_2Q.$$

Чтобы найти такие точки, подставим kP вместо Q и получим:

$$c_1P + d_1kP = (c_1 + d_1k)P = c_2P + d2kP = (c_2 + d_2k)P.$$

Это означает, что $(c_1 + d_1k)$ равно $(c_2 + d_2k)$ по модулю, равному количеству точек на кривой, которое не является секретом.

Отсюда

$$d_2k - d_1k = c_1 - c_2;$$
$$k(d_1 - d_2) = c_1 - c_2;$$
$$k = (c_1 - c_2)/(d_1 - d_2).$$

И мы нашли k, решив тем самым ECDLP.

Разумеется, это только общая картина – детали сложнее и интереснее. На практике эллиптические кривые строятся над полем по модулю длиной как минимум 256 бит, что делает атаки на эллиптическую криптографию с помощью попытки найти коллизию практически неосуществимыми, потому что для этого понадобилось бы порядка 2^{128} операций (в главе 6 мы определили стоимость нахождения коллизии в множестве 256-битовых чисел).

Протокол совместной выработки ключа Диффи–Хеллмана над эллиптическими кривыми

Напомним (см. главу 11), что в классическом протоколе Диффи–Хеллмана (DH) две стороны вырабатывают совместный секрет, обмениваясь несекретными значениями. Зная некоторое фиксированное число g, Алиса выбирает случайное секретное число a, вычисляет $A = g^a$, отправляет A Бобу, а Боб выбирает случайное секретное число b и отправляет $B = g^b$ Алисе. Затем каждая сторона объединяет свой секретный ключ с открытым ключом другой стороны, в результате чего получается одно и то же значение $A^b = B^a = g^{ab}$.

Вариант DH на эллиптической кривой совпадает с классическим DH с точностью до обозначений. В случае ECC Алиса, зная некоторую фиксированную точку G, выбирает случайное секретное число d_A, вычисляет $P_A = d_AG$ (произведение точки G на d_A) и отправляет P_A Бобу. Боб выбирает случайное секретное число d_B, вычисляет $P_B = d_BG$ и отправляет результат Алисе. Затем обе стороны вычисляют общий секрет $d_AP_B = d_BP_A = d_Ad_BG$. Этот метод называется *протоколом Диффи–Хеллмана на эллиптических кривых* (elliptic curve Diffie–Hellman – ECDH).

ECDH играет ту же роль для ECDLP, что DH для DLP: он безопасен при условии, что задача ECDLP трудная. Поэтому протоколы DH, опирающиеся на DLP, можно адаптировать для работы с эллиптическими кривыми. Например, DH с аутентификацией и протокол Менезеса–Кью–Вэнстоуна (MQV) будут безопасны и на эллиптических кривых. (На самом деле MQV первоначально и был определен над эллиптическими кривыми.)

Подписание с помощью эллиптических кривых

Стандартным алгоритмом цифровой подписи в ECC является *ECDSA* (elliptic curve digital signature algorithm). Он заменил подписи RSA и классические подписи DSA во многих приложениях. Например, это единственный алгоритм подписания в технологии биткойна, и он же поддерживается многими реализациями TLS и SSH.

Как и все схемы цифровой подписи, ECDSA состоит из алгоритма *генерирования подписи*, с помощью которого пользователь создает подпись, применяя свой закрытый ключ, и алгоритма *верификации*, позволяющего проверить действительность подписи, зная открытый ключ подписавшего. Подписывающая сторона хранит число d – закрытый ключ, а проверяющая – открытый ключ $P = dG$. Обе стороны заранее знают, какую эллиптическую кривую использовать, ее порядок (n, количество точек на кривой), а также координаты базовой точки G.

Генерирование подписи в ECDSA

Чтобы подписать сообщение, подписывающая сторона сначала вычисляет хеш сообщения h с помощью криптографической функции хеширования, например SHA-256 или BLAKE2; он интерпретируется как число от 0 до $n - 1$. Затем эта сторона выбирает случайное число k от 1 до $n - 1$ и вычисляет kG, точку с координатами (x, y). После этого подписывающая сторона полагает $r = x \bmod n$ и вычисляет $s = (h + rd)/k \bmod n$, а затем использует эти значения в качестве подписи, (r, s).

Длина подписи зависит от длин используемых координат. Например, если мы работаем с кривой, для которой координаты – 256-битовые числа, то r и s будут 256-битовыми, т. е. длина всей подписи составит 512 бит.

Верификация подписи в ECDSA

В алгоритме верификации используется открытый ключ подписывающей стороны для проверки действительности подписи.

Чтобы верифицировать подпись ECDSA (r, s) и хеш сообщения h, проверяющий сначала вычисляет величину $w = 1/s$, обратную подписи и равную $k/(h + rd) \bmod n$, т. к. s определено как $s = (h + rd)/k$. Затем проверяющий умножает w на h и находит u по следующей формуле:

$$wh = hk/(h + rd) = u.$$

Далее проверяющий умножает w на r и находит v:

$$wr = rk\,(h + rd) = v.$$

Зная u и v, проверяющий вычисляет точку Q:

$$Q = uG + vP.$$

Здесь P – открытый ключ подписывающей стороны, равный dG. Проверяющий принимает подпись, только если координата x точки Q равна значению r из подписи.

Эта процедура работает, потому что на последнем шаге мы вычисляем точку Q, подставляя вместо открытого ключа P его фактическое значение dG:

$$uG + vdG = (u + vd)G.$$

После подстановки вместо u и v их значений получаем

$$u + vd = hk/(h + rd) + drk/(h + rd) = (hk + drk)/(h + rd) = k(h + dr)/(h + rd) = k.$$

Таким образом, $(u + vd)$ равно значению k, выбранному при генерировании подписи, и, значит, $uG + vdG$ равно точке kG. Иными словами, алгоритму верификации удалось успешно вычислить точку kG – ту самую, что была вычислена на этапе генерирования подписи. Проверка считается завершенной, если проверяющая сторона убеждается, что координата x точки kG равна полученному значению r; в противном случае подпись отвергается как недействительная.

Сравнение подписей ECDSA и RSA

Эллиптическую криптографию часто рассматривают как альтернативу RSA в роли криптографии с открытым ключом, но у ECC и RSA мало общего. RSA применяется только для шифрования и подписания, тогда как ECC – это семейство алгоритмов, которые можно использовать для шифрования, генерирования подписей, совместной выработки ключа и более продвинутых криптографических функций, таких как личностное шифрование (вид шифрования, в котором ключи выводятся из персонального идентификатора, например адреса электронной почты).

При сравнении алгоритмов подписания RSA и ECC помните, что в RSA подписывающая сторона использует свой закрытый ключ d для вычисления подписи в виде $y = x^d \bmod n$, где x – подписываемые данные, а y – подпись. При верификации открытый ключ e используется для подтверждения того, что $y^e \bmod n$ равно x – очевидно, что этот процесс проще, чем в ECDSA.

Процедура верификации в RSA часто работает быстрее, чем генерирование подписи в ECC, потому что используется короткий открытый ключ e. Но у ECC есть два важных преимущества перед RSA:

более короткая подпись и скорость подписания. Поскольку ECC работает с меньшими числами, то и подпись получается короче, чем в RSA (сотни, а не тысячи бит), что, безусловно, существенно, если необходимо хранить или передавать много подписей. Подписание в ECDSA также гораздо быстрее, чем в RSA (хотя верификация подписи занимает примерно столько же времени), т. е. ECDSA для обеспечения сравнимого уровня безопасности нужны более короткие числа, чем RSA. Например, в листинге 12.2 показано, что ECDSA приблизительно в 150 раз быстрее при подписании и немного быстрее при верификации. Отметим, что и подписи в ECDSA короче, чем в RSA: 512 бит (два элемента по 256 бит) против 4096.

Листинг 12.2. Сравнение скорости 4096-битовых RSA-подписей с 256-битовыми подписями ECDSA

```
$ openssl speed ecdsap256 rsa4096
                              sign     verify     sign/s    verify/s
rsa 4096 bits            0.007267s  0.000116s      137.6      8648.0
                              sign     verify     sign/s    verify/s
256 bit ecdsa (nistp256)   0.0000s    0.0001s    21074.6      9675.7
```

Сравнивать производительность этих подписей разного размера справедливо, потому что они обеспечивают примерно одинаковую безопасность. Но на практике во многих системах используются RSA-подписи длиной 2048 бит, которые на несколько порядков менее безопасны, чем 256-битовая ECDSA-подпись. Благодаря меньшему размеру модуля 2048-битовая RSA быстрее 256-битовой ECDSA на этапе верификации, но все равно медленнее на этапе подписания, как явствует из листинга 12.3.

Листинг 12.3. Скорость 2048-битовых RSA-подписей

```
$ openssl speed rsa2048
                              sign     verify    sign/s   verify/s
rsa 2048 bits            0.000696s  0.000032s    1436.1    30967.1
```

Вывод: следует выбирать ECDSA, а не RSA, за исключением случаев, когда время верификации подписи критично *и* скорость подписания безразлична, как в ситуации, когда подпись генерируется один раз, а верифицируется много раз (например, исполняемое приложение Windows подписывается однократно, а верифицируется всеми исполняющими его системами).

Шифрование с помощью эллиптических кривых

Хотя эллиптическая криптография чаще используется для подписания, шифрование с ее помощью тоже возможно. Но на практике это делается редко из-за ограничений на размер открытого текста: обес-

печить такой же уровень безопасности, как для почти 4000 бит открытого текста в RSA, можно лишь для примерно 100 бит в ECC.

Простой способ шифрования с помощью эллиптических кривых – использовать *интегрированную схему шифрования* (integrated encryption scheme – IES), гибридный алгоритм с симметричными и асимметричными ключами, основанный на обмене ключами по протоколу Диффи–Хеллмана. По существу, IES шифрует сообщение путем генерирования пары ключей Диффи–Хеллмана, объединения закрытого ключа с открытым ключом получателя, формирования симметричного ключа по полученному разделяемому секрету и использования шифра с аутентификацией для шифрования сообщения.

При использовании совместно с эллиптическими кривыми IES полагается на трудность задачи ECDLP и называется *интегрированной схемой шифрования на эллиптических кривых* (elliptic-curve integrated encryption scheme – ECIES). Зная открытый ключ получателя, P, ECIES шифрует сообщение M следующим образом.

1. Выбрать случайное число d и вычислить точку $Q = dG$, где базовая точка G – фиксированный параметр. Здесь (d, Q) играет роль эфемерной пары ключей и используется только для шифрования M.
2. Вычислить разделяемый секрет ECDH, $S = dP$.
3. Использовать функцию формирования ключа (KDF) для формирования симметричного ключа K по S.
4. Зашифровать M с помощью K и симметричного шифра с аутентификацией, получив в результате шифртекст C и аутентификационный жетон T.

Таким образом, шифртекст в схеме ECIES состоит из эфемерного открытого ключа Q, за которым следуют C и T. Порядок дешифрирования очевиден: получатель вычисляет S, умножая R на свой закрытый показатель степени, а затем выводит ключ K, дешифрирует C и проверяет T.

Выбор кривой

Критерии для оценки безопасности эллиптической кривой включают порядок группы (т. е. число точек в ней), формулы сложения на кривой и происхождение.

Существует несколько типов эллиптических кривых, но не все они одинаково хороши для криптографических целей. Делая выбор, обращайте особое внимание на коэффициенты a и b в уравнении кривой $y^2 = x^3 + ax + b$; в противном случае можно получить небезопасную кривую. На практике обычно используется одна из де-факто стандартных кривых, но, зная, от чего зависит безопасность кривой, вы будете более осознанно подходить к выбору и оценке сопутствующих рисков. Вот о чем следует помнить.

- Порядок группы не должен быть произведением малых чисел, иначе решить задачу ECDLP окажется намного проще.
- В разделе «Сложение и умножение точек» выше мы узнали, что вычисление суммы $P + Q$ при $Q = P$ производится по специальной формуле. К сожалению, особая обработка этого случая может привести к утечке ценной информации, если противник сумеет отличить удвоение от сложения разных точек. Некоторые кривые являются безопасными *именно* потому, что для них все точки складываются по единой формуле. (Если для кривой не требуется специальная формула удвоения, то говорят, что кривая допускает *унифицированное* правило сложения.)
- Если авторы кривой не объясняют происхождение a и b, то их можно заподозрить в нечестной игре, потому что нет уверенности, что они специально не выбрали ослабленные значения, допускающие пока еще неизвестную атаку на криптосистему.

Рассмотрим некоторые из наиболее употребительных кривых, особенно используемые для создания подписей или в протоколе Диффи–Хеллмана.

Примечание *Дополнительные критерии и другие сведения о кривых можно найти на специально посвященном этому вопросу сайте https://safecurves.cr.yp.to/.*

Кривые, рекомендованные NIST

В 2000 году некоторые эллиптические кривые были стандартизованы NIST в документе FIPS 186 «Recommended Elliptic Curves for Federal Government Use». Пять кривых NIST по модулю простого числа называются *простыми кривыми*. Еще десять кривых работают с двоичными полиномами – математическими объектами, благодаря которым аппаратная реализация особенно эффективна. (Мы не будем здесь говорить о двоичных полиномах, потому что они редко используются в сочетании с эллиптическими кривыми.)

Наиболее употребительны простые кривые NIST. Из них самой популярной является кривая P-256 над полем по модулю 256-битового простого числа $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. Кривая P-256 описывается уравнением $y^2 = x^3 - 3x + b$, где b – 256-битовое число. NIST также рекомендует кривые над полем $\mathbf{Z}_p$, где p имеет длину 192, 224, 384 и 521 бит.

Кривые NIST иногда критикуют за то, что только сам их создатель, АНБ, знает о происхождении коэффициента b в уравнениях. Единственное объяснение, которое нам дают, – что b является результатом хеширования кажущейся случайной константы с помощью функции SHA-1. Например, параметр b кривой P-256 происходит от константы `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`.

Никто не знает, почему АНБ выбрало именно эту константу, но большинство специалистов не верят, что в происхождении кривых скрыта какая-то слабость.

Кривая Curve25519

Дэниель Дж. Бернштейн явил миру кривую Curve25519 (произносится «кривая двадцать-пять-пять-девятнадцать») в 2006 году. Стремясь добиться производительности, он спроектировал Curve25519 так, чтобы она была быстрее и позволяла использовать более короткие ключи, чем стандартные кривые. Но у Curve25519 есть также преимущества в плане безопасности, потому что, в отличие от кривых NIST, в ней нет никаких подозрительных констант, а для сложения различных точек и удвоения точки применяется одна и та же унифицированная формула.

Уравнение кривой Curve25519, $y^2 = x^3 + 486662x^2 + x$, немного отличается от других уравнений, встречающихся в этой главе, но она тем не менее принадлежит семейству эллиптических кривых. А необычная форма дает возможность применить специальные приемы, благодаря чему программная реализация Curve25519 оказывается быстрой.

Curve25519 определена над полем по модулю $2^{255} - 19$, это 256-битовое простое число, максимально близкое к 2^{255}. Коэффициент $b = 486662$ – наименьшее число, удовлетворяющее критерию безопасности, сформулированному Бернштейном. В совокупности эти особенности делают Curve25519 более заслуживающей доверия, чем кривые NIST с их непонятно откуда взявшимися коэффициентами.

Curve25519 используется всюду: в Google Chrome, в системах Apple, в OpenSSH и много где еще. Однако она не стандартизована NIST, а некоторые приложения предпочитают придерживаться стандартов.

Примечание *Если хотите узнать все подробности и аргументацию в пользу Curve25519, посмотрите презентацию Дэниеля Дж. Бернштейна «The first 10 years of Curve25519» 2016 года, доступную по адресу http://cr.yp.to/talks.html#2016.03.09/.*

Другие кривые

На момент написания этой книги в большинстве криптографических приложений используются кривые NIST или Curve25519, но есть и устаревшие стандарты, а также новые кривые, продвигаемые внутри комитетов по стандартизации. Из старых национальных стандартов упомяну французские кривые ANSSI и немецкие Brainpool: два семейства, которые не поддерживают унифицированные формулы сложения и используют константы неизвестного происхождения.

Некоторые из новых кривых более эффективны, чем старые, и свободны от подозрений; они предлагают разные уровни безопасности и оптимизации. Из примеров отмечу Curve41417, вариант Curve25519, работающий с более длинными числами и обеспечивающий более высокий уровень безопасности (приблизительно 200 бит); Ed448-Goldilocks, 448-битовую кривую, впервые предложенную в 2014 году и претендовавшую на роль стандарта интернета; а также шесть кривых, предложенных в работе Aranha et al. «A note on high-security general-

purpose elliptic curves» (см. *http://eprint.iacr.org/2013/647/*), хотя они используются редко. Подробное описание всех этих кривых выходит за рамки данной книги.

Какие возможны проблемы

Недостатки эллиптических кривых связаны со сложностью и большой поверхностью атаки. Поскольку у них больше параметров, чем у классического протокола Диффи–Хеллмана, открывается больше возможностей для ошибок и неправильного использования, а возможно, и для программных дефектов в реализации. Программное обеспечение на основе эллиптических кривых может быть уязвимо к атакам по побочным каналам из-за больших чисел, участвующих в арифметических операциях. Если скорость вычислений зависит от входных данных, то противник может получить информацию о формулах, используемых при шифровании.

В следующих разделах я разберу два примера уязвимостей, которым подвержены эллиптические кривые, даже если реализация безопасна. Это уязвимости протокола, а не реализации.

ECDSA с недостаточной случайностью

Подписание ECDSA рандомизировано, поскольку в вычислении $s = (h + rd)/k \bmod n$ участвует секретное случайное число k. Однако если одно и то же k повторно используется для подписания второго сообщения, то противник мог бы объединить два значения, $s_1 = (h_1 + rd)/k$ и $s_2 = (h_2 + rd)/k$, получить $s_1 - s_2 = (h_1 - h_2)/k$, затем и $k = (h_1 - h_2)/(s_1 - s_2)$. Если k известно, то закрытый ключ d легко восстанавливается:

$$(ks_1 - h_1)/r = ((h_1 + rd) - h_1)/r = rd/r = d.$$

В отличие от RSA-подписей, которые не позволят восстановить ключ, если используется слабый генератор псевдослучайных чисел (PRNG), в ECDSA недостаточно случайные числа могут привести к раскрытию k, как случилось в атаке на игровую консоль PlayStation 3 в 2010 году, представленной группой fail0verflow на 27-м конгрессе хакеров Chaos Communication Congress в Берлине.

Взлом ECDH с помощью другой кривой

ECDH можно элегантно взломать, если пренебречь проверкой входных точек. Основная причина заключается в том, что формулы для вычисления координат суммы точек $P + Q$ не содержат коэффициента b, а зависят только от координат P и Q и коэффициента a (в случае удвоения точки). У этого факта есть печальное следствие – при сложении двух точек никогда нельзя быть уверенным, что вы работаете с правильной кривой, а не складываете точки, лежащие на кривой

с другим коэффициентом b. Это означает, что ECDH можно взломать, как описано в следующем сценарии, называемом *атакой с неправильной кривой*.

Предположим, что Алиса и Боб выполняют протокол ECDH и согласовали кривую и базовую точку G. Боб отправляет свой открытый ключ $d_B G$ Алисе. Алиса, вместо того чтобы отправить открытый ключ $d_A G$ в соответствии с ранее согласованной кривой, отправляет точку на другой кривой, случайно или намеренно. К несчастью, эта кривая слабая, что позволяет Алисе выбрать точку P, для которой решить задачу ECDLP легко. Она выбирает точку низкого порядка, для которой существует сравнительно небольшое k такое, что $kP = O$.

Теперь Боб, думая, что располагает настоящим открытым ключом, вычисляет то, что, по его мнению, является разделяемым секретом $d_B P$, хеширует его и использует получившийся ключ, чтобы зашифровать данные, отправляемые Алисе. Проблема в том, что когда Боб вычисляет $d_B P$, он, ничего не подозревая, использует более слабую кривую. А в результате, поскольку P выбрана из малой подгруппы гораздо большей группы точек, $d_B P$ также будет принадлежать этой малой подгруппе, что позволит противнику, знающему порядок P, эффективно определить разделяемый секрет $d_B P$.

Предотвратить это можно, проверив, что точки P и Q принадлежат правильной кривой, для чего нужно убедиться, что их координаты удовлетворяют ее уравнению. В этом случае мы будем работать только с точками на безопасной кривой и сорвем атаку.

Эта атака с неправильной кривой была найдена в 2015 году для некоторых реализаций протокола TLS, в которых протокол ECDH использовался для выработки сеансовых ключей. (Детали см. в работе Jager, Schwenk, and Somorovsky «Practical Invalid Curve Attacks on TLS-ECDH».)

Для дополнительного чтения

Эллиптическая криптография – увлекательная и сложная тема, требующая привлечения серьезной математики. Я не стал обсуждать такие важные понятия, как порядок точки, кофактор кривой, проективные координаты, точки кручения и методы решения задачи ECDLP. Читатели с математическим складом ума могут найти информацию по этим и другим вопросам в книге Cohen and Frey «Handbook of Elliptic and Hyperelliptic Curve Cryptography» (Chapman and Hall/CRC, 2005). Обзор Bos, Halderman, Heninger, Moore, Naehrig, and Wustrow «Elliptic Curve Cryptography in Practice» 2013 года также содержит хорошее введение с иллюстрациями и практическими примерами (*https://eprint.iacr.org/2013/734/*).

13

ПРОТОКОЛ TLS

Протокол *Transport Layer Security* (*TLS*), известный также под названием *Secure Socket Layer* (*SSL*), хотя так назывался его предшественник, лежит в основе безопасности в интернете. TLS защищает соединения между серверами и клиентами, будь то соединение между веб-сайтом и его посетителями, почтовыми серверами и клиентами, мобильным приложением и его серверами или серверами видеоигр и игроками. Без TLS не было бы безопасной онлайновой торговли, безопасного онлайнового банкинга, да и вообще ничего безопасного в онлайне.

TLS не зависит от приложения; он ничего не знает о типе шифруемых данных. Это означает, что мы можем использовать его для веб-приложений, работающих по протоколу HTTP, а также в любой системе, где клиентский компьютер или устройство инициируют сеанс с удаленным сервером. Например, TLS широко применяется для межмашинной связи в приложениях интернета вещей (IoT).

В этой главе я дам краткий обзор TLS. Вы увидите, что с годами TLS заметно усложнился. К сожалению, сложность и разрастание

функциональности принесли с собой уязвимости, а найденные в запутанной реализации дефекты получили широкое освещение в прессе – вспомните об уязвимостях Heartbleed, BEAST, CRIME и POODLE, затронувших миллионы веб-серверов.

В 2013 году инженеры, уставшие исправлять все новые криптографические уязвимости в TLS, решили полностью переработать его и начали работу над версией TLS 1.3. Как вы узнаете из этой главы, из TLS 1.3 были исключены ненужные и небезопасные функции, а старые алгоритмы заменены современными шифрами. В результате получился более простой, быстрый и безопасный протокол.

Но прежде чем рассказывать о том, как работает TLS 1.3, рассмотрим, какую проблему вообще пытается решить TLS и почему она существует.

Целевые приложения и требования

Наибольшую известность TLS принесло то, что он отвечает за букву *S* в сайтах, работающих по протоколу HTTPS, и, стало быть, за значок замка в адресной строке браузера, сообщающий, что страница защищена. Основным побудительным мотивом для создания TLS было желание обезопасить работу с приложениями электронной коммерции и онлайнового банкинга путем шифрования соединений с веб-сайтом, что позволило бы защитить номера кредитных карт, учетные данные пользователей и другую конфиденциальную информацию.

TLS также помогает защищать связь в интернете в целом, потому что организует *безопасный канал* между клиентом и сервером, гарантирующий, что передаваемые данные конфиденциальны, подлинны и не модифицированы.

Одна из целей безопасности TLS – предотвратить атаки с человеком посередине, когда противник перехватывает зашифрованный трафик от передающей стороны, расшифровывает его, получает открытый текст, а затем снова зашифровывает и отправляет принимающей стороне. TLS препятствует атакам с человеком посередине благодаря аутентификации серверов (и факультативно клиентов) с помощью сертификатов и доверенных удостоверяющих центров, которые мы подробнее обсудим в разделе «Сертификаты и удостоверяющие центры» ниже.

Чтобы надеяться на широкое распространение, TLS должен удовлетворять дополнительным требованиям: эффективность, интероперабельность, расширяемость и универсальность.

В контексте TLS эффективность означает минимизацию накладных расходов по сравнению с незашифрованными соединениями. Это хорошо и для сервера (чтобы снизить стоимость оборудования на стороне поставщика услуг), и для клиентов (чтобы избежать ощутимых задержек или сокращения срока эксплуатации аккумуляторов мобильных устройств). Протокол должен быть интероперабельным, т. е. работать на любом оборудовании и в любой операционной си-

стеме. Он должен быть расширяемым, чтобы поддерживать дополнительные функции и алгоритмы. И наконец, он должен быть универсальным, т. е. не привязанным к какому-то конкретному приложению (это роднит его с протоколом TCP, которому также безразлично, какой прикладной протокол работает поверх него).

Набор протоколов TLS

Чтобы защитить клиент-серверные коммуникации, TLS состоит из разных версий нескольких протоколов, образующих *набор* протоколов TLS. И хотя акроним *TLS* расшифровывается как *Transport Layer Security*, это на самом деле не протокол транспортного уровня. Обычно TLS расположен между транспортным протоколом TCP и протоколом прикладного уровня, например HTTP или SMTP, и защищает данные, передаваемые по TCP-соединению.

TLS может также работать поверх транспортного протокола UDP, который используется для передачи данных без установления соединения, например голоса или видео. Но, в отличие от TCP, UDP не гарантирует ни доставку, ни правильный порядок пакетов. Поэтому UDP-версия TLS несколько отличается и называется *DTLS* (*Datagram Transport Layer Security*). Дополнительные сведения о TCP и UDP см. в книге Charles Kozierok «The TCP/IP Guide» (No Starch Press, 2005).

Семейство протоколов TLS и SSL: краткая история

Жизнь TLS началась в 1995 году, когда компания Netscape, разработчик одноименного браузера, создала предшественника TLS, протокол Secure Socket Layer (SSL). SSL был далек от совершенства, и в обеих версиях SSL 2.0 и SSL 3.0 были дефекты, относящиеся к безопасности. Так что пользоваться SSL нельзя ни в коем случае, следует применять только TLS, но это лишь усиливает неразбериху, потому что TLS часто называют «SSL» даже профессионалы в области безопасности.

Более того, не все версии TLS безопасны. TLS 1.0 (1999) – наименее безопасная версия, хотя все-таки безопаснее, чем SSL 3.0. TLS 1.1 (2006) лучше, но включает ряд алгоритмов, которые сегодня считаются нестойкими.

TLS 1.2 (2008) еще лучше, но эта версия сложна и обеспечивает высокую безопасность только при условии правильной настройки (а это не простое дело). Кроме того, ее сложность повышает риск наличия дефектов в реализациях и риск задания неправильной конфигурации. Например, TLS 1.2 поддерживает AES в режиме CBC, который уязвим к атакам на оракул дополнения.

Версия TLS 1.2 унаследовала множество функций и проектных решений от прежних версий, поэтому не оптимальна с точки зрения безопасности и производительности. Чтобы расчистить эти завалы, инженеры-криптографы изобрели TLS заново, оставив только хоро-

шее и добавив средства безопасности. Итогом их работы стала версия TLS 1.3, из которой удалено все лишнее, а то, что осталось, сделано более безопасным, эффективным и простым. По сути дела, TLS 1.3 – это зрелый TLS.

TLS в двух словах

TLS состоит из двух основных протоколов: один определяет, как передавать данные, другой – какие данные передавать. *Протокол записи* определяет формат пакета для инкапсуляции данных протоколов верхних уровней и передачи их другой стороне. Это простой протокол, и нередко даже забывают, что он является частью TLS.

Протокол подтверждения связи, или просто *квитирование*, – это протокол выработки ключа в TLS. Нередко ошибочно считают, что это и есть протокол TLS, но на самом деле протоколы квитирования и подтверждения связи неразделимы.

Процедура квитирования инициируется клиентом, желающим установить безопасное соединение с сервером. Клиент посылает начальное сообщение ClientHello, содержащее параметры, в т. ч. шифр, который хочет использовать клиент. Сервер проверяет это сообщение и его параметры и отправляет в ответ сообщение ServerHello. После того как клиент и сервер обработали сообщения друг друга, они готовы приступить к обмену данными, зашифрованными ключами, которые были выработаны в ходе выполнения протокола подтверждения связи, описанного ниже.

Сертификаты и удостоверяющие центры

Самый важный шаг процедуры квитирования в TLS и основа безопасности этого протокола – *проверка сертификата*, когда сервер предъявляет клиенту *сертификат*, подтверждающий его подлинность.

Сертификат – это, по существу, открытый ключ, дополненный подписью данного ключа и сопутствующей информацией (в т. ч. доменным именем). Например, при подключении к сайту *https://www.google.com/* ваш браузер получает от некоторого сервера в сети сертификат, содержащий сведения типа «я *google.com*, и мой открытый ключ равен [*key*]», и проверяет подпись этого сертификата. Если подпись успешно верифицирована, то говорят, что сертификат (и его открытый ключ) *надежный*, и браузер может продолжить подключение. (О цифровых подписях см. главы 10 и 12.)

Откуда браузер знает открытый ключ, необходимый для верификации подписи? Тут-то и возникает концепция *удостоверяющего центра* (*УЦ*) – *доверенной третьей стороны*, гарантирующей, что открытые ключи в сертификатах действительно принадлежат сайтам или организациям, которые заявляют права на них. Не будь УЦ, мы не смогли бы проверить, что открытый ключ в сертификате от *google.com* принадлежит Google, а не противнику, организовавшему атаку с человеком посередине.

Например, команда в листинге 13.1 показывает, что происходит при использовании пакета OpenSSL для инициирования TLS-подключения к *www.google.com* по сетевому порту 443, который предназначен для обмена данными по протоколу HTTP, защищенному TLS (т. е. HTTPS).

Листинг 13.1. Установление TLS-подключения к www.google.com и получение сертификатов для аутентификации этого подключения

```
$ openssl s_client -connect www.google.com:443
CONNECTED(00000003)
--опущено--
---
Certificate chain
❶ 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
  i:/C=US/O=Google Inc/CN=Google Internet Authority G2
❷ 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
  i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
❸ 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
  i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIEgDCCA2igAwIBAgIISCr6QCbz5rowDQYJKoZIhvcNAQELBQAwSTELMAkGA1UE
BhMCVVMxEzARBgNVBAoTCkdvb2dsZSBJbmMxJTAjBgNVBAMTHEdvb2dsZSBJbnRl
--опущено--
cb9reU8in8yCaH8dtzrFyUracpMureWnBeajOYXRPTdCFccejAh/xyH5SKDOOZ4v
3TP9GBtClAH1mSXoPhX73dp7jipZqgbY4kiEDNx+hformTUFBDHD0eO/s2nqwuWL
pBH6XQ==
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
issuer=/C=US/O=Google Inc/CN=Google Internet Authority G2
--опущено--
```

Я оставил только интересную часть вывода, а именно сам сертификат. Обратите внимание, что до первого сертификата (начинающегося строкой `BEGIN CERTIFICATE`) расположено описание *цепочки сертификатов*, в которой строка, начинающаяся с `s:`, описывает имя субъекта, а строка, начинающаяся с `i:`, – владельца подписи. Здесь сертификат `0` выдан *google.com* ❶, сертификат `1` ❷ принадлежит организации, подписавшей сертификат `0`, а сертификат `2` ❸ – организации, подписавшей сертификат `1`. Организация, выпустившая сертификат `1` (GeoTrust), предоставила организации Google Internet Authority право выпуска сертификата (именно сертификата `0`) для доменного имени *www.google.com*, делегировав тем самым доверие к Google Internet Authority.

Очевидно, что сами удостоверяющие центры должны быть достойны доверия и выпускать сертификаты только для заслуживающих доверия организаций и что они должны защищать свои закрытые ключи, чтобы никакой противник не мог выпустить сертификаты от их лица (например, с целью притвориться легитимным сервером *google.com*).

Чтобы узнать, что находится внутри сертификата, выполним команду, показанную в листинге 13.2, и скопируем первый сертификат из листинга 13.1.

Листинг 13.2. Декодирование сертификата, полученного от www.google.com

```
$ openssl x509 -text -noout
-----BEGIN CERTIFICATE-----
--опущено--
-----END CERTIFICATE-----
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 5200243873191028410 (0x482afa4026f3e6ba)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2
        Validity
            Not Before: Dec 15 14:07:56 2016 GMT
            Not After : Mar 9 13:35:00 2017 GMT
        Subject: C=US, ST=California, L=Mountain View, O=Google Inc, CN=www.
google.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:bc:bc:b2:f3:1a:16:3b:c6:f6:9d:28:e1:ef:8e:
                    92:9b:13:b2:ae:7b:50:8f:f0:b4:e0:36:8d:09:00:
--опущено--
                    8f:e6:96:fe:41:41:85:9d:a9:10:9a:09:6e:fc:bd:
                    43:fa:4d:c6:a3:55:9a:9e:07:8b:f9:b1:1e:ce:d1:
                    22:49
                Exponent: 65537 (0x10001)
--опущено--
    Signature Algorithm: sha256WithRSAEncryption
        94:cd:66:55:83:f1:16:7d:46:d8:66:21:06:ec:c6:9d:7c:1c:
        2b:c1:f6:4f:b7:3e:cd:01:ad:69:bd:a1:81:6a:7c:96:f5:9c:
--опущено--
        85:fa:2b:99:35:05:04:31:c3:d1:e3:bf:b3:69:ea:c2:e5:8b:
        a4:11:fa:5d
```

Здесь мы видим результат декодирования командой `openssl x509` сертификата, представленного блоком текста в кодировке base64. Поскольку OpenSSL знает структуру этого блока, он может сказать, что находится внутри сертификата, в т. ч. серийный номер, информацию о версии, идентифицирующую информацию, срок действия (строки `Not Before` (не раньше) и `Not After` (не позже)), открытый ключ (в данном случае модуль и открытый показатель степени RSA) и подпись всего вышеперечисленного.

Хотя специалисты по безопасности и криптографы часто заявляют, что вся система сертификатов изначально порочна, это одно из лучших имеющихся в нашем распоряжении решений – наряду с полити-

кой доверия при первом использовании (trust-on-first-use – TOFU)), принятой в SSH.

Протокол записи

Все данные, которыми стороны обмениваются по протоколу TLS 1.3, передаются в виде последовательности *записей TLS*, т. е. пакетов данных. Протокол записи TLS (*уровень записи*) – это, по сути дела, транспортный протокол, ничего не знающий о семантике транспортируемых данных; именно в силу этой особенности TLS пригоден для любых приложений.

Впервые протокол записи TLS используется для передачи данных в процессе квитирования. После того как связь подтверждена и обе стороны знают секретный ключ, данные приложения разбиваются на порции, которые передаются в составе записей TLS.

Структура записи TLS

Запись TLS – это блок данных размером не более 16 килобайт, имеющий следующую структуру.

- Первый байт представляет тип передаваемых данных, он равен 22 для данных квитирования, 23 для зашифрованных данных и 21 для уведомлений. В спецификации TLS 1.3 это значение называется ContentType (тип содержимого).
- Второй и третий байты равны соответственно 3 и 1. Они фиксированы по историческим причинам и не являются уникальной особенностью версии TLS 1.3. В спецификации это 2-байтовое значение называется ProtocolVersion (версия протокола).
- Четвертый и пятый байты кодируют длину передаваемых данных 16-битовым числом, которое не может быть больше 2^{14} (16 КБ).
- Остальные байты – это передаваемые данные (*полезная нагрузка*), их длина равна значению, закодированному в четвертом и пятом байтах записи.

Примечание *Запись TLS имеет сравнительно простую структуру. Как мы видели, заголовок записи состоит всего из трех полей. Для сравнения пакет протокола IPv4 содержит 14 полей, предшествующих полезной нагрузке, а в сегменте TCP таких полей 13.*

Если первый байт записи TLS 1.3 (ContentType) равен 23, то полезная нагрузка зашифрована и аутентифицирована с помощью шифра с аутентификацией. Полезная нагрузка состоит из шифртекста и аутентификационного жетона, которые принимающая сторона должна будет дешифрировать. Но откуда получатель знает тип шифра и ключ? Это обеспечивает TLS: если вы получили зашифрованную запись TLS, значит, уже знаете шифр и ключ, потому что они были согласованы в процессе выполнения протокола подтверждения связи.

Одноразовые числа

В отличие от многих других протоколов, например Encapsulating Security Payload (ESP), являющегося частью IPsec, в записях TLS нет одноразового числа, необходимого шифру с аутентификацией.

Одноразовые числа, применяемые для шифрования и дешифрирования записей TLS, выводятся из 64-битовых порядковых номеров, которые локально хранятся каждой стороной и увеличиваются на единицу для каждой новой записи. Клиент, зашифровывающий данные, выводит одноразовое число, выполняя XOR между порядковым номером и значением `client_write_iv`, которое само выводится из разделяемого секрета. Сервер поступает аналогично, но использует другое значение, `server_write_iv`.

Например, если передается три записи TLS, то для первой записи одноразовое число будет равно 0, для второй 1, а для третьей 2; при получении этих трех записей также будут использованы одноразовые числа 0, 1, 2, именно в таком порядке. Повторное применение одинаковых порядковых номеров для шифрования передаваемых и дешифрирования принимаемых данных не является слабостью протокола, потому что они объединяются операцией XOR с разными константами (`client_write_iv` и `server_write_iv`) и потому что в каждом направлении применяются разные секретные ключи.

Дополнение нулями

Записи TLS 1.3 поддерживают интересное свойство, которое называется *дополнением нулями* и уменьшает риск атак посредством анализа трафика. *Анализ трафика* – это метод, с помощью которого противник извлекает полезную информацию из трафика, ориентируясь на хронометраж, объем переданных данных и т. д. Например, поскольку шифртекст обычно мало отличается по размеру от открытого текста, даже при использовании стойкого шифрования противник может определить приблизительный размер сообщения, просто глядя на длину соответствующего ему шифртекста.

Дополнение нулями означает добавление нулей к открытому тексту, чтобы увеличить размер шифртекста и тем самым заставить наблюдателя думать, что зашифрованное сообщение длиннее, чем на самом деле.

Протокол подтверждения связи

Подтверждение связи, или *квитирование*, – это основной протокол согласования в TLS; в процессе его выполнения клиент и сервер вырабатывают общие секретные ключи для инициирования безопасной связи. Во время квитирования клиент и сервер играют разные роли. Клиент предлагает какие-то конфигурации (версию TLS и набор шифров в порядке предпочтения), а сервер выбирает, какую конфигурацию он будет использовать. Сервер стремится уважать пожелания клиента, но не обязан это делать. Чтобы обеспечить интероперабель-

ность реализаций и гарантировать, что любой сервер, реализующий TLS 1.3, сможет прочитать данные, отправленные любым клиентом, реализующим TLS 1.3 (даже если используются разные библиотеки или языки программирования), в спецификации TLS 1.3 описан также формат отправляемых данных.

На рис. 13.1 показан обмен данными в процессе квитирования в соответствии со спецификацией TLS 1.3. Как видим, клиент отправляет серверу сообщение «я хочу установить с тобой TLS-соединение. Вот какие шифры я поддерживаю для шифрования TLS-записей, а вот мой открытый ключ Диффи–Хеллмана». Открытый ключ должен быть сгенерирован специально для этого сеанса TLS, и клиент хранит у себя ассоциированный с ним закрытый ключ. Отправленное клиентом сообщение включает также 32-байтовое случайное значение и факультативную информацию (дополнительные параметры и все такое). Первое сообщение называется *ClientHello*, это последовательность байтов определенного формата, описанного в спецификации TLS 1.3.

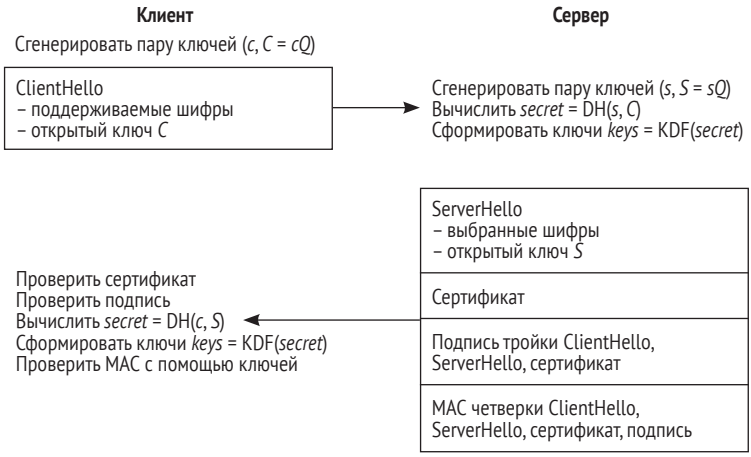

Рис. 13.1. Процесс подтверждения связи в TLS 1.3 при подключении к HTTPS-сайту

Отметим, что в спецификации описано также, в каком формате следует отправлять данные, чтобы гарантировать интероперабельность реализаций.

Сервер получает сообщение ClientHello, проверяет, что оно правильно отформатировано, и посылает в ответ сообщение *ServerHello*. В этом сообщении куча информации: шифр для шифрования записей TLS, открытый ключ Диффи–Хеллмана, 32-байтовое случайное значение (обсуждается в разделе «Защита от понижения версии»), сертификат, подпись, вычисленная по всей информации в сообщениях ClientHello и ServerHello (с применением ключа, ассоциированного с открытым ключом сертификата), имитовставка (MAC), вычисленная по той же информации плюс подпись. MAC вычисляется с применением симметричного ключа, сформированного по разделяемому сек-

рету Диффи–Хеллмана, который сервер вычисляет на основе закрытого ключа Диффи–Хеллмана и открытого ключа клиента.

Получив сообщение ServerHello, клиент проверяет действительность сертификата, вычисляет разделяемый секрет Диффи–Хеллмана, формирует на его основе симметричные ключи и проверяет имитовставку, отправленную сервером. Если все проверки завершились успешно, то клиент готов к отправке зашифрованных сообщений серверу.

Примечание *Заметим, однако, что TLS 1.3 поддерживает много параметров и расширений, поэтому может вести себя иначе, чем описано здесь (и показано на рис. 13.1). Например, можно настроить протокол квитирования TLS 1.3 так, что сервер будет требовать от клиента сертификат, чтобы удостоверить его подлинность. TLS 1.3 также поддерживает предварительно разделенные ключи.*

Посмотрим, как все это выглядит на практике. Допустим, мы развернули TLS 1.3, чтобы обеспечить безопасный доступ к сайту *https://www.nostarch.com/*. Если зайти на него в браузере, то браузер отправит серверу сайта сообщение ClientHello, включающее перечень поддерживаемых им шифров. Сайт ответит сообщением ServerHello, содержащим сертификат, который включает открытый ключ, ассоциированный с доменом *www.nostarch.com*. Клиент проверит действительность сертификата, прибегнув к помощи одного из удостоверяющих центров, прописанных в браузере (полученный сертификат должен быть подписан надежным удостоверяющим центром, сертификат которого должен быть включен в хранилище сертификатов браузера, иначе проверка не пройдет). После того как все проверки завершатся успешно, браузер запросит начальную страницу от сервера *www.nostarch.com*.

После успешного завершения квитирования весь обмен сообщениями между клиентом и сервером зашифрован и аутентифицирован. Подслушивающий противник может узнать, что клиент с данным IP-адресом общается с сервером с другим IP-адресом, и может наблюдать зашифрованные сообщения, но не сможет ни получить открытый текст, ни изменить зашифрованные сообщения (иначе принимающая сторона заметила бы, что данными манипулируют, поскольку сообщения не только зашифрованы, но и аутентифицированы). Для большинства приложений такой безопасности достаточно.

Криптографические алгоритмы TLS 1.3

Мы знаем, что TLS 1.3 пользуется алгоритмами шифрования с аутентификацией, функцией формирования ключей (хеш-функцией, которая выводит ключи из разделяемого секрета), а также операцией Диффи–Хеллмана. Но как именно все это работает, какие алгоритмы применяются и насколько они безопасны?

Что касается выбора шифров с аутентификацией, то TLS 1.3 поддерживает только три алгоритма: AES-GCM, AES-CCM (этот режим чуть

менее эффективен, чем GCM) и потоковый шифр ChaCha20 в сочетании с имитовставкой Poly1305 (определенной в RFC 7539). Поскольку TLS 1.3 не позволяет задать небезопасную длину ключа, например 64 или 80 бит (то и другое слишком мало), то секретный ключ может иметь длину 128 бит (AES-GCM или AES-CCM) или 256 бит (AES-GCM или ChaCha20-Poly1305).

Функция формирования ключа (KDF) на рис. 13.1 основана на HKDF, построении на основе HMAC (см. главу 7), определенном в RFC 5869; в нем используется хеш-функция SHA-256 или SHA-384.

Варианты выполнения операции Диффи–Хеллмана (на основе протокола квитирования в TLS 1.3) ограничены эллиптической криптографией и мультипликативной группой целых чисел по простому модулю (как в традиционном протоколе Диффи–Хеллмана). Но нельзя использовать любую эллиптическую кривую или группу: поддерживаются только три кривые NIST, а также кривые Curve25519 (обсуждалась в главе 12) и Curve448, определенные в RFC 7748. TLS 1.3 поддерживает еще протокол DH над группой целых чисел, а не над эллиптическими кривыми. Конкретно поддерживается пять групп, определенных в RFC 7919: с модулями длиной 2048, 3072, 4096, 6144 и 8192 бит.

2048-битовая группа, вероятно, является самым слабым звеном TLS 1.3. Считается, что она обеспечивает уровень безопасности меньше 100 бит, тогда как остальные варианты дают по меньшей мере 128-битовую безопасность. Поэтому решение о поддержке 2048-битовой группы можно расценить как не согласованное с остальными проектными решениями TLS 1.3.

Улучшения TLS 1.3 по сравнению с TLS 1.2

Протокол TLS 1.3 очень сильно отличается от своего предшественника. Прежде всего он избавился от слабых алгоритмов типа MD5, SHA-1, RC4 и AES в режиме CBC. Кроме того, если в TLS 1.2 записи часто защищались с помощью комбинации шифра и MAC (например, HMAC-SHA-1) в рамках построения MAC-затем-шифрование, то TLS 1.3 поддерживает только более эффективные и безопасные шифры с аутентификацией. TLS 1.3 также отказался от согласования кодировки точек эллиптической кривой и определяет единственный формат точек для всех кривых.

Одной из главных целей разработки TLS 1.3 было исключение из версии 1.2 функциональности, ослаблявшей протокол, и уменьшение сложности, а значит, и поверхности атаки. Например, из TLS 1.3 исключено факультативное сжатие данных, из-за которого стала возможной атака CRIME на TLS 1.2. В этой атаке использовался тот факт, что длина сжатого сообщения раскрывает информацию о его содержимом.

Однако в TLS 1.3 включены также новые возможности, позволяющие сделать соединение более безопасным или более эффективным.

Я разберу три из них: защиту от понижения версии, квитирование с одним периодом кругового обращения и возобновление сеанса.

Защита от понижения версии

В TLS 1.3 *защита от понижения версии* предназначена для противостояния атакам, при которых противник вынуждает клиента и сервер использовать более слабую версию TLS, чем 1.3. Чтобы организовать такую атаку, противник перехватывает и модифицирует сообщение ClientHello, сообщая серверу, что клиент не поддерживает TLS 1.3. Теперь противник может эксплуатировать уязвимости в прежних версиях TLS.

Пытаясь защититься от атак с понижением версии, сервер TLS 1.3 использует три типа паттернов в 32-байтовом случайном значении, отправляемом в сообщении ServerHello, чтобы идентифицировать тип запрошенного соединения. Паттерн должен соответствовать запросу клиентом конкретного типа TLS-соединения. Получив неверный паттерн, клиент понимает, что что-то пошло не так.

А именно если клиент запрашивает соединение с версией TLS 1.2, то первые восемь из 32 байт равны 44 4F 57 4E 47 52 44 01, а если он запрашивает соединение с версией TLS 1.1, то они равны 44 4F 57 4E 47 52 44 00. Если же клиент запрашивает соединение с версией TLS 1.3, то эти восемь байт должны быть случайными. Например, если клиент отправил сообщение ClientHello с запросом TLS 1.3, но противник модифицировал его, запросив TLS 1.1, то, получив в ответ сообщение ServerHello, клиент увидит неправильный паттерн и будет знать, что сообщение ClientHello было модифицировано. (Противник не может по своему усмотрению модифицировать случайное 32-байтовое значение, выбранное сервером, потому что оно криптографически подписано.)

Квитирование с одним периодом кругового обращения

В типичной процедуре квитирования TLS 1.2 клиент отправляет какие-то данные серверу, ждет ответа, затем отправляет еще данные, снова ждет ответа и только потом начинает отправлять зашифрованные сообщения. Задержка составляет два периода кругового обращения (round-trip time – RTT). Что же касается TLS 1.3, то период кругового обращения только один, как показано на рис. 13.1. Сэкономить удается сотни миллисекунд. На первый взгляд, мало, но если вспомнить, что серверы популярных служб обрабатывают тысячи соединений в секунду, то набегает вполне прилично.

Возобновление сеанса

TLS 1.3 работает быстрее, чем 1.2, но его можно сделать еще быстрее (на несколько сотен миллисекунд), полностью исключив периоды кругового обращения, предшествующие зашифрованной части сеан-

са. Трюк состоит в том, чтобы использовать *возобновление сеанса* – метод, при котором общий ключ, уже выработанный клиентом и сервером в предыдущем сеансе, используется для инициализации нового сеанса. Возобновление сеанса дает два важных преимущества: клиент может сразу же начать отправку зашифрованных данных, и в возобновленных сеансах нет нужды использовать сертификаты.

На рис. 13.2 показано, как работает возобновление сеанса. Сначала клиент отправляет сообщение ClientHello, которое включает идентификатор уже разделенного с сервером ключа (обозначен *PSK*, от *pre-shared key* – предварительно разделенный ключ), а также новый открытый ключ DH. Клиент также может включить в первое сообщение зашифрованные данные (они называются *данными на нулевом периоде кругового обращения*). Отвечая на сообщение ClientHello, сервер включает имитовставку, вычисленную по отправленным данным. Клиент проверяет имитовставку и понимает, что взаимодействует с тем же сервером, что и раньше, поэтому проверять сертификат излишне. Клиент и сервер выполняют протокол выработки ключа Диффи–Хеллмана, как при нормальном квитировании, и последующие сообщения шифруются ключами, зависящими как от PSK, так и от вновь вычисленного разделяемого секрета Диффи–Хеллмана.

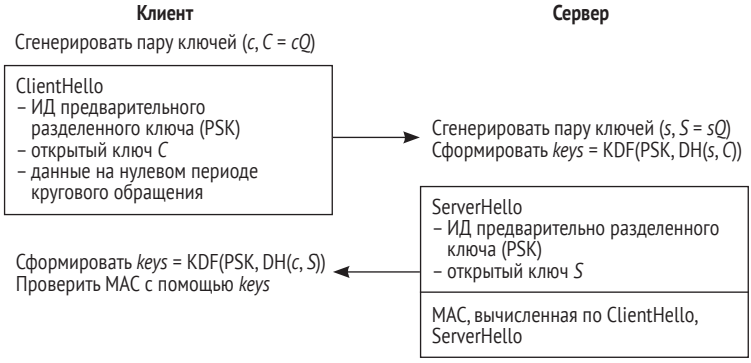

Рис. 13.2. Квитирование при возобновлении сеанса в TLS 1.3. Данные на нулевом периоде кругового обращения посылаются в составе сообщения ClientHello

Стойкость TLS

Мы оценим стойкость TLS 1.3 относительно двух главных аспектов безопасности, рассмотренных в главе 11: аутентификации и секретности прошлого.

Аутентификация

В процессе квитирования TLS 1.3 сервер аутентифицирует себя клиенту с помощью механизма сертификатов. Однако клиент не аутентифицирован, хотя и может аутентифицировать себя в серверном

приложении (например, Gmail), включив имя пользователя и пароль в запись, TLS, следующую за квитированием. Если клиент уже установил сеанс с удаленной службой, то может аутентифицироваться, отправив *безопасный кук*, который разрешено посылать только по TLS-соединению.

В некоторых случаях клиенты могут аутентифицироваться с помощью основанного на сертификатах механизма, аналогичного тому, что использует сервер: клиент отправляет серверу *клиентский сертификат*, а сервер проверяет его, перед тем как что-то разрешить клиенту. Однако клиентские сертификаты используются редко, потому что они усложняют работу и клиентов, и сервера (стороны, выпускающей сертификаты): клиенты должны выполнить сложную последовательность операций, чтобы включить сертификат в свою систему и защитить его закрытый ключ, а выпускающая сторона должна среди прочего гарантировать, что сертификат получили только авторизованные клиенты.

Секретность прошлого

Напомним, что протокол совместной выработки ключа обеспечивает секретность прошлого, если предыдущие сеансы остаются нескомпрометированными в случае компрометации текущего сеанса. В модели утечки данных компрометируются только временные секреты, а в модели вскрытия – долговременные секреты.

К счастью, секретность прошлого в TLS 1.3 не боится ни утечки, ни вскрытия. В случае модели утечки данных противник добывает временные секреты, например сеансовые ключи или закрытые ключи Диффи–Хеллмана для конкретного сеанса (значения *c*, *s*, *secret* и *keys* на рис. 13.1). Однако эти значения он может использовать только для дешифрирования данных в текущем сеансе, но не в предыдущих, потому что там действовали другие значения *c* и *s* (а значит, и другие ключи).

В модели вскрытия противник получает также долговременные секреты (закрытый ключ, соответствующий открытому ключу в сертификате). Однако это помогает при дешифрировании предыдущих сеансов ничуть не больше, чем знание временных секретов, потому что закрытый ключ применяется только для аутентификации сервера, так что безопасность прошлого снова устояла.

Но что может произойти на практике? Предположим, что противник скомпрометировал машину клиента и получил доступ ко всей ее памяти. Теперь противник может восстановить сеансовые ключи и секреты TLS для текущего сеанса из памяти. Но еще важнее другое – если предыдущие ключи все еще хранятся в памяти, то противник сможет найти и их тоже и использовать для дешифрирования предыдущих сеансов, обойдя тем самым теоретическую безопасность прошлого. Поэтому, чтобы реализация TLS действительно гарантировала безопасность прошлого, необходимо стирать ключи в памяти после окончания использования, обычно их просто обнуляют.

Какие возможны проблемы

TLS 1.3 отвечает требованиям, предъявляемым к безопасному протоколу связи общего назначения, но он не является непробиваемым. Как и любая система безопасности, при определенных обстоятельствах он может пасть (например, если предположения проектировщиков о реальных атаках оказались неверными). К сожалению, даже самую последнюю версию TLS 1.3, сконфигурированную с самыми стойкими шифрами, все-таки можно скомпрометировать. Например, безопасность TLS 1.3 опирается на предположение, что все три стороны (клиент, сервер и удостоверяющий центр) ведут себя честно, но что, если одна сторона скомпрометирована или сама реализация TLS неудовлетворительна?

Скомпрометированный удостоверяющий центр

Корневыми удостоверяющими центрами (корневыми УЦ) называются организации, которым браузеры доверяют при проверке сертификатов удаленных серверов. Например, при проверке сертификата, предъявленного сайтом *www.google.com*, предполагается, что легитимность его владельца верифицирована доверенным УЦ. Браузер проверяет сертификат, верифицируя подпись выпустившей его стороны. Поскольку только УЦ знает закрытый ключ, необходимый для создания этой подписи, мы предполагаем, что больше никто не может выпустить подлинные сертификаты от имени этого УЦ. Очень часто сертификат сайта подписывается не корневым УЦ, а промежуточным, который связан с корневым цепочкой сертификатов.

Но предположим, что закрытый ключ УЦ скомпрометирован. Теперь противник сможет использовать закрытый ключ УЦ для создания сертификата любого URL в домене *google.com* без одобрения Google. И что тогда? Противник сможет использовать эти сертификаты, чтобы притвориться владельцем легитимного сервера или поддомена, скажем *mail.google.com*, и перехватывать учетные данные пользователя и все его сообщения. Именно так и случилось в 2011 году, когда противник сумел взломать сеть голландского удостоверяющего центра DigiNotar и создал сертификаты, которые выглядели законными сертификатами, выпущенными DigiNotar. Тогда это было использовано для выпуска поддельных сертификатов нескольких служб Google.

Скомпрометированный сервер

Если сервер скомпрометирован и полностью контролируется противником, то потеряно всё: противник будет видеть все передаваемые данные еще до того, как они зашифрованы, и все принимаемые данные после того, как они дешифрированы. Он также сможет захватить закрытый ключ сервера, что позволит ему замаскировать свой вредоносный сервер под легитимный. Очевидно, что в этом случае TLS не спасет.

По счастью, подобные катастрофы редко случаются в таких известных приложениях, как Gmail или iCloud, которые хорошо защищены и часто хранят закрытые ключи в отдельном модуле безопасности. Атаки на уязвимости веб-приложений, например с внедрением запросов к базе данных или межсайтового скриптинга, более распространены, поскольку не зависят от безопасности TLS и выполняются по легитимному TLS-соединению. Такие атаки могут скомпрометировать имена и пароли пользователей и т. п.

Скомпрометированный клиент

TLS-безопасность окажется под угрозой и тогда, когда удаленный противник скомпрометировал клиента, например браузер. Добившись в этом успеха, противник сможет захватить сеансовые ключи, читать любые дешифрированные данные и т. д. Он даже сможет установить сертификат фальшивого УЦ в браузер клиента и таким образом заставить браузер безропотно принимать недействительные сертификаты и перехватывать контроль над TLS-соединениями.

Большая разница между скомпрометированным сервером или УЦ и скомпрометированным клиентом заключается в том, что в последнем случае страдает только один, а не *все* клиенты.

Дефекты реализации

Как и любая криптографическая система, TLS может вести себя неправильно, если в реализации имеются дефекты. Ходячий пример дефектов TLS – атака Heartbleed (см. рис. 13.3), вызывающая переполнение буфера в реализации второстепенной функции TLS – контрольном сигнале (heartbeat) – в пакете OpenSSL. Heartbleed была обнаружена в 2014 году независимо сотрудником Google и компанией Codenomicon. Она затронула миллионы серверов и клиентов TLS.

Как показано на рис. 13.3, клиент сначала отправляет буфер и его длину серверу, чтобы проверить, работает ли сервер. В этом примере в буфере находится строка *BANANAS*, и клиент явно сообщает, что длина этого слова – семь букв. Сервер читает семибуквенное слово и возвращает его клиенту.

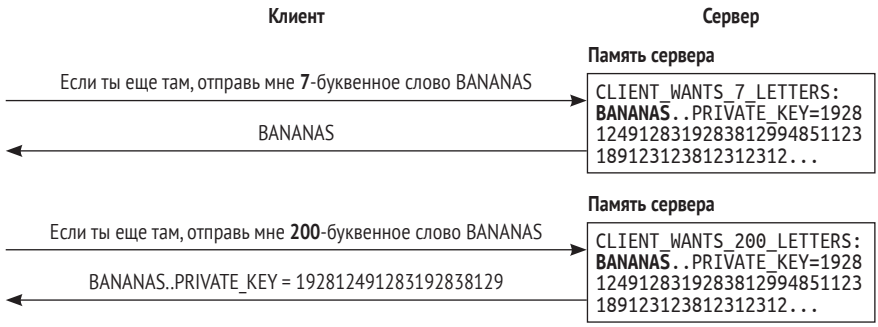

Рис. 13.3. Ошибка Heartbleed в реализации TLS в OpenSSL

Проблема в том, что сервер не проверяет, что длина правильна, и пытается прочитать столько символов, сколько указал клиент. Следовательно, если клиент задаст длину, бóльшую, чем реальная длина строки, то сервер прочитает из памяти слишком много данных и вернет их клиенту, хотя дополнительные данные могут содержать конфиденциальную информацию, скажем закрытые ключи или сеансовые куки.

Неудивительно, что известие о дефекте Heartbleed стало шоком. Чтобы избежать таких ошибок в будущем, разработчики OpenSSL и других распространенных реализаций TLS теперь организовали строгий критический анализ кода и пользуются автоматизированными инструментами, в частности фаззерами, для выявления потенциальных проблем.

Для дополнительного чтения

В самом начале я сказал, что эту главу не стоит рассматривать как полное руководство по TLS и что имеет смысл поглубже покопаться в TLS 1.3. Начать можно со спецификации TLS 1.3, которая содержит все, что нужно знать о протоколе (кроме разве что мотивов его создания). Ее можно найти на домашней странице рабочей группы TLS Working Group (TLSWG) по адресу *https://tlswg.github.io/*.

Кроме того, хочу упомянуть о двух важных инициативах, касающихся TLS.

- Тест SSL Labs (*https://www.ssllabs.com/ssltest/*) – бесплатная служба, предоставляемая компанией Qualys, которая проверяет конфигурацию TLS браузера или сервера и возвращает рейтинг безопасности, а также рекомендации по улучшению. Если вы сами настраиваете свой TLS-сервер, воспользуйтесь этим тестом – рейтинг «A» подтверждает его безопасность.
- Let's Encrypt (*https://letsencrypt.org/*) – некоммерческая организация, которая предлагает службу «автомагического» развертывания TLS на ваших HTTP-серверах. Она включает средства автоматического генерирования сертификатов и конфигурирования TLS-сервера, при этом поддерживаются все сколько-нибудь распространенные веб-серверы и операционные системы.

14

КВАНТОВАЯ И ПОСТКВАНТОВАЯ КРИПТОГРАФИЯ

Предыдущие главы были посвящены сегодняшнему состоянию криптографии, а в этой мы рассмотрим ее будущее на горизонте, скажем, ста или более лет – когда появятся *квантовые компьютеры*. Это компьютеры, в которых для выполнения алгоритмов, отличающихся от привычных нам, используются квантово-механические явления. Пока что квантовых компьютеров не существует, и кажется, что построить их очень трудно, но если когда-то они появятся, то потенциально смогут взломать криптографию на базе RSA, протокола Диффи–Хеллмана и эллиптических кривых, т. е. всю криптографию с открытым ключом, которая стандартизована и развернута сегодня.

Чтобы застраховаться от рисков, которые несут с собой квантовые компьютеры, криптографы разработали альтернативные алго-

ритмы с открытым ключом, получившие название *постквантовых*. В 2015 году АНБ призвало переходить на квантовостойкие алгоритмы, безопасные даже при использовании квантовых компьютеров, а в 2017 году американский институт NIST, отвечающий за стандарты, начал процесс, который в конечном итоге должен привести к стандартизации постквантовых алгоритмов.

Эта глава содержит неформальный обзор принципов работы квантовых компьютеров, а также начальные сведения о постквантовых алгоритмах. Нам понадобится немного математики, но ничего более сложного, чем основы арифметики и линейной алгебры, так что не пугайтесь – незнакомых обозначений не будет.

Как работают квантовые компьютеры

В модели квантовых вычислений используется квантовая физика, что позволяет организовывать вычисления по-другому и делать такие вещи, на которые неспособны обычные компьютеры, например эффективно взламывать криптосистемы на основе RSA и эллиптических кривых. Но квантовый компьютер – это вовсе не сверхбыстрый обычный компьютер. На самом деле квантовые компьютеры не могут решить любую проблему, слишком трудную для классического компьютера, например поиск полным перебором или **NP**-полные задачи.

Квантовые компьютеры основаны на квантовой механике, разделе физики, изучающем поведение субатомных частиц, для которого характерна истинная случайность. В отличие от классических компьютеров, оперирующих битами, которые принимают значение 0 или 1, квантовые компьютеры имеют дело с *квантовыми битами* (или *кубитами*), которые могут быть одновременно равны 0 и 1, – это состояние неопределенности называется *суперпозицией*. Физики открыли, что в микромире частицы, например электроны и фотоны, ведут себя совсем не так, как подсказывает интуиция: до момента наблюдения электрон не занимает определенное место в пространстве, а может находиться сразу в нескольких местах (т. е. в состоянии суперпозиции). Но стоит начать наблюдать за ним – эта операция в квантовой физике называется *измерением*, – как он замирает в фиксированном, но случайном положении и больше не находится в состоянии суперпозиции. Именно эта магия квантового мира и позволяет создавать кубиты в квантовом компьютере.

Но квантовые компьютеры работают только потому, что существует еще сильнее сводящее с ума явление, *запутанность*: две частицы можно соединить (запутать) таким образом, что наблюдение одной даст значение другой, даже если частицы находятся на большом удалении друг от друга (несколько километров или даже световых лет). Это поведение иллюстрируется *парадоксом Эйнштейна–Подольского–Розена (ЭПР)*, из-за которого Эйнштейн поначалу отвергал квантовую механику. (См. углубленное изложение причин в статье по адресу *https://plato.stanford.edu/entries/qt-epr/*.)

Чтобы лучше объяснить, как работает квантовый компьютер, мы должны отличать собственно квантовый компьютер (оборудование, состоящее из квантовых битов) от квантовых алгоритмов (работающее на нем программное обеспечение, состоящее из *квантовых вентилей*).

Квантовые биты

Квантовые биты (кубиты) или их группы характеризуются числовыми *амплитудами*, которые близки к вероятностям, но не являются ими. Если вероятность – это число от 0 до 1, то амплитуда – комплексное число вида $a + b \times i$, или просто $a + bi$, где a и b – вещественные числа, а i – *мнимая единица*. Число i служит для образования *мнимых чисел* вида bi, где b – вещественное число. При умножении вещественного числа на i мы получаем мнимое число, а умножение i на себя дает –1, т. е. $i^2 = -1$.

В отличие от вещественных чисел, которые можно рассматривать как прямую (см. рис. 14.1), *комплексные числа* можно интерпретировать как плоскость (двумерную), как показано на рис. 14.2. Здесь ось x соответствует части a представления $a + bi$, ось y – части b, а штриховые линии – вещественной и мнимой частям комплексного числа. Так, длина отрезка вертикальной прямой от точки $3 + 2i$ до 3 равна двум единицам (2 – величина мнимой части $2i$).

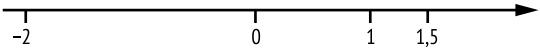

Рис. 14.1. Представление вещественных чисел точками на бесконечной прямой

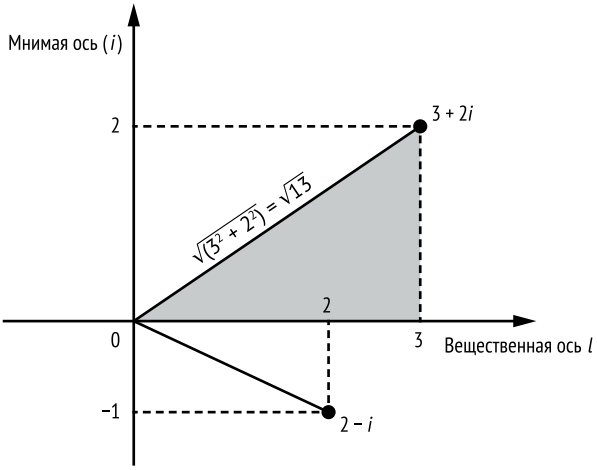

Рис. 14.2. Представление комплексных чисел точками на двумерной плоскости

По рис. 14.2 видно, что для вычисления длины отрезка, соединяющего начало координат (0) с точкой $a + bi$, можно воспользоваться тео-

ремой Пифагора, если рассмотреть его как гипотенузу прямоугольного треугольника. Эту длину, равную квадратному корню из суммы квадратов координат точки, $\sqrt{a^2 + b^2}$, мы называем *модулем* комплексного числа $a + bi$ и обозначаем $|a + bi|$.

В квантовом компьютере регистры состоят из одного или нескольких кубитов в состоянии суперпозиции, которое характеризуется множеством комплексных чисел. Но, как мы увидим, эти комплексные числа – амплитуды – не могут быть произвольными.

Амплитуды одного кубита

Один кубит характеризуется двумя амплитудами, которые я буду обозначать α (альфа) и β (бета). Тогда состояния кубита можно записать в виде $\alpha|0\rangle + \beta|1\rangle$, где нотация $|\ \rangle$. Это значит, что при наблюдении данного кубита он окажется равным 0 с вероятностью $|\alpha|^2$ и 1 с вероятностью $|\beta|^2$. Конечно, чтобы можно было говорить о вероятностях, $|\alpha|^2$ и $|\beta|^2$ должны быть числами от 0 до 1 такими, что $|\alpha|^2 + |\beta|^2 = 1$.

Например, предположим, что имеется кубит Ψ (пси) с амплитудами $\alpha = 1/\sqrt{2}$ и $\beta = 1/\sqrt{2}$. Это можно записать в виде

$$\Psi = (1/\sqrt{2})|0\rangle + (1/\sqrt{2})|1\rangle = (|0\rangle + |1\rangle)/\sqrt{2}.$$

Эта запись означает, что в кубите Ψ значение 0 имеет амплитуду $1/\sqrt{2}$ и значение 1 такую же амплитуду. Чтобы получить по амплитудам настоящую вероятность, нужно вычислить модуль числа $1/\sqrt{2}$ (он равен $1/\sqrt{2}$, потому что мнимая часть отсутствует), а затем возвести его в квадрат: $(1/\sqrt{2})^2 = 1/2$. То есть если мы пронаблюдаем кубит Ψ, то с вероятностью 1/2 увидим 0 и с такой же вероятностью 1.

Теперь рассмотрим кубит Φ (фи)

$$\Phi = (i/\sqrt{2})|0\rangle - (1/\sqrt{2})|1\rangle = (i|0\rangle - |1\rangle)/\sqrt{2} \text{ или } |\Phi\rangle = (i/\sqrt{2}, -1/\sqrt{2}).$$

Кубит Φ принципиально отличается от Ψ, потому что амплитуды Ψ равны, а амплитуды Φ различны: $\alpha = i/\sqrt{2}$ (положительное мнимое число) и $\beta = -1/\sqrt{2}$ (отрицательное вещественное число). Но при наблюдении Φ мы увидим амплитуды 0 и 1 с такой же вероятностью 1/2, как в случае Ψ. Действительно, вероятность увидеть 0 равна

$$|\alpha|^2 = \left(\sqrt{\left(1/\sqrt{2}\right)^2}\right)^2 = 1/\sqrt{2}^2 = 1/2.$$

Примечание *Поскольку $\alpha = i/\sqrt{2}$, его можно записать в виде $a + bi$ с $a = 0$ и $b = 1/\sqrt{2}$, так что вычисление модуля дает $|\alpha| = \sqrt{a^2 + b^2} = 1/\sqrt{2}$.*

Вывод: разные кубиты могут вести себя одинаково с точки зрения наблюдателя (вероятность увидеть 0 для обоих кубитов одинакова), но при этом иметь разные амплитуды. Это означает, что вероятности увидеть 0 или 1 не полностью характеризуют кубит. Можно провести

аналогию с тенью предмета на стене, форма которой дает представление о ширине и высоте, но не позволяет судить о глубине. В случае кубитов этим скрытым измерением является значение амплитуды. Положительно оно или отрицательно? Является вещественным числом или мнимым?

Примечание *Чтобы упростить обозначения, кубит часто записывают просто в виде пары амплитуд* (α, β). *Наш предыдущий пример можно записать так:* $|\Psi\rangle = (1/\sqrt{2}, 1/\sqrt{2})$.

Амплитуды группы кубитов

Мы рассмотрели одиночные кубиты, но что можно сказать о нескольких кубитах? Например, *квантовый байт* можно образовать из 8 кубитов, организованных так, что их квантовые состояния каким-то образом связаны друг с другом (говорят, что кубиты запутаны, это сложное физическое явление). Такой квантовый байт можно описать следующим образом, где α – амплитуды каждого из 256 возможных значений группы из 8 кубитов:

$$\alpha_0|00000000\rangle + \alpha_1|00000001\rangle + \alpha_2|00000010\rangle + \alpha_3|00000011\rangle + \ldots + \alpha_{255}|11111111\rangle.$$

Заметим, что сумма $|\alpha_0|^2 + |\alpha_1|^2 + \ldots + |\alpha_{255}|^2$ должна быть равна 1, поскольку это сумма вероятностей.

Нашу группу из 8 кубитов можно рассматривать как множество $2^8 = 256$ амплитуд, потому что она имеет 256 возможных конфигураций, каждая со своей амплитудой. Но в реальности у нас было бы всего восемь, а не 256 физических объектов. 256 амплитуд – это неявная характеристика группы из 8 кубитов; каждое из этих 256 чисел может принимать бесконечно много значений. Обобщая, можно сказать, что группа из 8 кубитов характеризуется множеством 2^n комплексных чисел, и их количество экспоненциально возрастает с ростом числа кубитов.

Кодирование экспоненциально большого множества комплексных чисел высокой точности – основная причина, из-за которой классический компьютер не может смоделировать квантовый; для хранения того же количества информации, которое содержится всего в *n* кубитах, потребовалось бы невообразимо много памяти (порядка 2^n).

Квантовые вентили

Понятия амплитуды и квантовых вентилей уникальны для квантовых вычислений. Там, где классический компьютер использует регистры, память и микропроцессор для выполнения последовательности команд, квантовый компьютер производит обратимое преобразование группы кубитов с помощью серии квантовых вентилей, а затем измеряет значение одного или нескольких кубитов. Квантовые компьютеры обещают увеличение вычислительной мощности, потому что,

имея всего лишь n кубитов, они могут обрабатывать 2^n чисел (амплитуд кубитов). У этого свойства имеются весьма важные следствия.

С математической точки зрения, квантовые алгоритмы представляют собой цепь *квантовых вентилей*, которая преобразует множество комплексных чисел (амплитуд), перед тем как произвести завершающее измерение значений одного или нескольких кубитов (см. рис. 14.3). Иногда квантовые алгоритмы называют также квантовыми *вентильными матрицами* или *квантовыми схемами*.

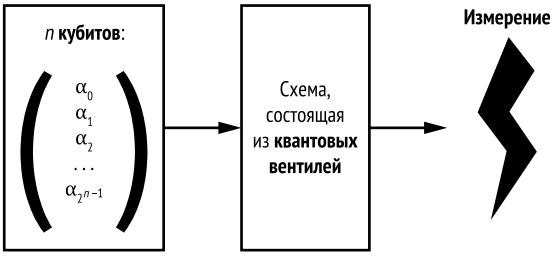

Рис. 14.3. Принцип работы квантового алгоритма

Квантовые вентили как операции умножения матриц

В отличие от булевых логических элементов в классическом компьютере (AND, XOR и т. д.), квантовый вентиль воздействует на группу амплитуд, как при умножении матрицы на вектор. Например, чтобы применить простейший квантовый вентиль *равнозначности* к кубиту Ф, мы рассматриваем I как матрицу 2×2 и умножаем ее на вектор-столбец, состоящий из двух амплитуд Ф:

$$I|\Phi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1\times\frac{1}{\sqrt{2}} + 0\times\left(-\frac{1}{\sqrt{2}}\right) \\ 0\times\frac{1}{\sqrt{2}} + 1\times\left(-\frac{1}{\sqrt{2}}\right) \end{pmatrix} = \begin{pmatrix} i/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = |\Phi\rangle.$$

Результатом такого умножения матрицы на вектор является другой вектор-столбец, состоящий из двух элементов, первый из которых равен скалярному произведению первой строки матрицы I на входной вектор (результат прибавления первых элементов 1 и $i/\sqrt{2}$ к произведению вторых элементов 0 и $-1/\sqrt{2}$), и аналогично для второго значения.

Примечание *На практике квантовый компьютер не стал бы явно вычислять произведение матрицы на вектор, потому что матрицы были бы слишком большими. (Потому-то квантовый компьютер и невозможно смоделировать на классическом.) Вместо этого квантовый компьютер применил бы к кубитам как физическим частицам преобразование, эквивалентное умножению на матрицу. Не понятно? Так не зря же Ричард Фейнман сказал: «Если вы думаете, что понимаете квантовую механику, значит, вы ее не понимаете».*

Квантовый вентиль Адамара

До сих пор мы видели только квантовый вентиль равнозначности *I*, который практически бесполезен, потому что ничего не делает и оставляет кубит неизменным. А теперь познакомимся с одним из самых полезных квантовых вентилей – *вентилем Адамара*, который обычно обозначается *H* и определяется следующим образом (обратите внимание на знак минус в правой нижней позиции):

$$H = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}.$$

Посмотрим, что произойдет, если применить этот вентиль к кубиту $|\Psi\rangle = (1/\sqrt{2}, 1/\sqrt{2})$:

$$H|\Psi\rangle = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}\begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 + 1/2 \\ 1/2 - 1/2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0.$$

Применив вентиль Адамара *H* к $|\Psi\rangle$, мы получим кубит $|0\rangle$, для которого значение $|0\rangle$ имеет амплитуду 1, а значение $|1\rangle$ амплитуду 0. Это означает, что кубит будет вести себя детерминировано, т. е. при наблюдении этого кубита мы всегда будем видеть 0 и никогда 1. Иными словами, случайность начального кубита $|\Psi\rangle$ утрачена.

А что, если еще раз применить вентиль Адамара к кубиту $|0\rangle$?

$$H|0\rangle = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = |\Psi\rangle.$$

Это вернет нас к кубиту $|\Psi\rangle$ и рандомизированному состоянию. Действительно, вентиль Адамара часто используется в квантовых алгоритмах для перехода от детерминированного состояния к случайному равномерно распределенному.

Не все матрицы являются квантовыми вентилями

Хотя квантовые вентили можно рассматривать как умножение на матрицу, не каждой матрице соответствует квантовый вентиль. Напомню, что кубит состоит из комплексных чисел α и β, амплитуды которых должны удовлетворять условию $|\alpha|^2 + |\beta|^2 = 1$. Если после умножения кубита на матрицу получаются две амплитуды, не удовлетворяющие этому условию, то результат не является кубитом. Квантовым вентилям соответствуют только матрицы, сохраняющие свойство $|\alpha|^2 + |\beta|^2 = 1$, они называются *унитарными*.

Унитарные матрицы (и, по определению, квантовые вентили) *обратимы*, т. е., зная результат операции, мы можем вычислить исходный кубит путем умножения на *обратную* матрицу. По этой причине говорят, что квантовые вычисления – вид *обратимых вычислений*.

Квантовое ускорение

Квантовое ускорение имеет место, когда с помощью квантового компьютера задачу можно решить быстрее, чем с помощью классического. Например, для поиска элемента в неупорядоченном списке из n элементов на классическом компьютере в среднем необходимо выполнить $n/2$ операций, потому что придется просмотреть все элементы, предшествующие искомому. (В среднем элемент найдется после просмотра половины списка.) Никакой классический алгоритм не сможет добиться лучшего результата. Однако существует квантовый алгоритм поиска элемента примерно за $\sqrt{n}$ операций, что на несколько порядков лучше, чем $n/2$. Например, если n равно 1 000 000, то $n/2$ равно 500 000, а $\sqrt{n}$ – всего 1000.

Мы пытаемся измерить разницу между квантовыми и классическими алгоритмами в терминах *временной сложности* в нотации $O()$. В примере выше квантовый алгоритм работает за время $O(\sqrt{n})$, а классический не может быть быстрее, чем $O(n)$. Такое различие во временной сложности называется *квадратичным ускорением*. Даже этот результат выглядит очень неплохо, но можно добиться гораздо большего.

Экспоненциальное ускорение и задача Саймона

Экспоненциальное ускорение – золотая мечта квантовых вычислений. Так бывает, когда задача, требующая экспоненциального времени при решении на классическом компьютере, например $O(2^n)$, может быть решена за полиномиальное время на квантовом компьютере, т. е. за время $O(n^k)$ для некоторого фиксированного k. Такое ускорение превращает практически неразрешимую задачу в разрешимую. (Напомню, что криптографы и специалисты по теории сложности считают задачи с экспоненциальным временем неразрешимыми, а с полиномиальным – практически осуществимыми.)

Каноническим примером экспоненциального ускорения является *задача Саймона*. В этой задаче функция $\mathbf{f}()$ преобразует n-битовые строки в n-битовые строки так, что выход $\mathbf{f}()$ выглядит случайно с одной оговоркой: существует такое значение m, что для любых двух значений x, y, удовлетворяющих условию $\mathbf{f}(x) = \mathbf{f}(y)$, имеет место равенство $y = x \oplus m$. Требуется найти m.

Классический алгоритм решения задачи Саймона сводится к нахождению коллизии, что требует приблизительно $2^{n/2}$ обращений к $\mathbf{f}()$. Квантовый же алгоритм, показанный на рис. 14.4, может решить ее примерно за n обращений, т. е. его временная сложность составляет всего $O(n)$, что крайне эффективно.

Здесь мы инициализируем $2n$ кубитов значением $|0\rangle$, применяем вентили Адамара (H) к первым n кубитам, затем применяем вентиль $Q_\mathbf{f}$ к обеим группам по n кубитов. Получив две группы по n кубитов, x и y, вентиль $Q_\mathbf{f}$ преобразует квантовое состояние $|x\rangle|y\rangle$ в состояние $|x\rangle|\mathbf{f}(x) \oplus y\rangle$. То есть он вычисляет функцию $\mathbf{f}()$ на квантовом состоя-

нии обратимым образом, потому что мы можем перейти от нового состояния к старому, вычислив $\mathbf{f}(x)$, а затем применив XOR к результату и $\mathbf{f}(x) \oplus y$. (К сожалению, объяснение того, почему все это работает, выходит за рамки книги.)

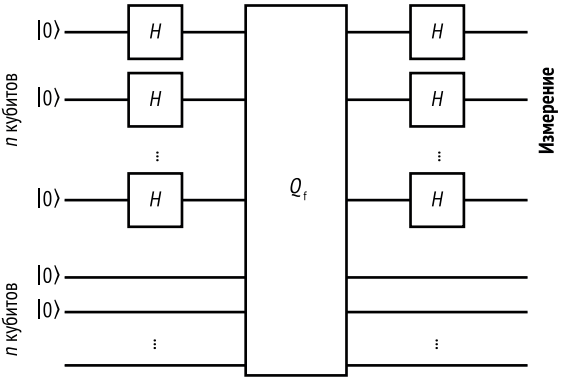

Рис. 14.4. Схема квантового алгоритма, эффективно решающего задачу Саймона

Экспоненциальное ускорение для задачи Саймона можно использовать против симметричных шифров только в очень специальных случаях, но в следующем разделе мы рассмотрим применение квантовых вычислений, обещающее стать настоящим убийцей криптографии.

Угроза со стороны алгоритма Шора

В 1995 году сотрудник компании AT&T Питер Шор опубликовал отрезвляющую статью, озаглавленную «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer»[1]. *Алгоритм Шора* – это квантовый алгоритм, который приводит к экспоненциальному ускорению при решении задач факторизации, дискретного логарифмирования (DLP) и дискретного логарифмирования на эллиптических кривых (ECDLP). На классическом компьютере эти задачи неразрешимы, но на квантовом их вполне можно решить. А следовательно, квантовый компьютер можно использовать для вскрытия любого криптографического алгоритма, зависящего от сложности этих задач, включая RSA, протокол Диффи–Хеллмана, эллиптическую криптографию и все применяемые в настоящее время криптографические механизмы. Иными словами, безопасность RSA или эллиптической криптографии оказывается не выше, чем у шифра Цезаря. (Шор мог с тем же успехом назвать свою статью «Взлом всей криптографии с открытым ключом на квантовом компьютере».)

[1] «Алгоритмы с полиномиальным временем работы для решения задач факторизации и дискретного логарифмирования на квантовом компьютере». – *Прим. перев.*

Известный специалист по теории сложности Скотт Ааронсон назвал алгоритм Шора «одним из выдающихся научных достижений конца XX века».

На самом деле алгоритм Шора решает более общий класс задач, чем факторизация и дискретное логарифмирование. А именно если функция $\mathbf{f}()$ *периодическая*, т. е. существует такое ω (период), что $\mathbf{f}(x + \omega) = \mathbf{f}(x)$ для любого x, то алгоритм Шора эффективно находит ω. (Это звучит похоже на рассмотренную выше задачу Саймона, и действительно алгоритм Саймона стал главным источником вдохновения для Шора.) Способность алгоритма Шора эффективно вычислять период функции важна для криптографов, потому что может быть использована для атаки на криптографию с открытым ключом, как станет ясно из дальнейшего.

Обсуждение того, как именно алгоритм Шора достигает экспоненциального ускорения, содержало бы слишком много технических деталей для этой книги, но ниже я покажу, как можно использовать его для атаки на криптографию с открытым ключом, а именно для решения задач факторизации и дискретного логарифмирования (см. главу 9), которые лежат в основе RSA и протокола Диффи–Хеллмана.

Решение задачи факторизации с помощью алгоритма Шора

Пусть требуется разложить на множители большое число $N = pq$. Сделать это будет легко, если мы сможем вычислить период $a^x \bmod N$; эта задача трудна для классического компьютера, но легко решается на квантовом. Сначала выбираем случайное число a, меньшее N, и просим алгоритм Шора найти период ω функции $\mathbf{f}(x) = a^x \bmod N$. Если период найден, то мы имеем $a^x \bmod N = a^{x+\omega} \bmod N$ (т. е. $a^x \bmod N = a^x a^\omega \bmod N$), а это означает, что $a^\omega \bmod N = 1$, или $a^\omega - 1 \bmod N = 0$. Иначе говоря, $a^\omega - 1$ кратно N, т. е. $a^\omega - 1 = kN$ для некоторого неизвестного числа k.

Ключевое наблюдение – тот факт, что легко можно разложить число $a^\omega - 1$ в произведение двух множителей: $a^\omega - 1 = (a^{\omega/2} - 1)(a^{\omega/2} + 1)$. Затем можно вычислить наибольший общий делитель $a^{\omega/2} - 1$ и N и проверить, получен ли нетривиальный множитель N (т. е. значение, отличное от 1 и N). Если нет, тот же алгоритм можно выполнить повторно с другим значением a. После нескольких попыток мы найдем множитель N. Таким образом, мы восстановили закрытый ключ RSA по открытому, что позволяет дешифрировать сообщения или подделывать подписи.

Но насколько легко проделать это вычисление? Заметим, что лучший классический алгоритм факторизации N работает за время, экспоненциально зависящее от n, длины N в битах (т. е. $n = \log_2 N$). А алгоритм Шора работает за время, *полиномиально* зависящее от n, а точнее за время $O(n^2(\log n)(\log\log n))$. Это означает, что будь у нас квантовый компьютер, мы смогли бы выполнить алгоритм Шора и получить ре-

зультат за разумное время (может быть, дни, недели или месяцы), а не за тысячи лет.

Алгоритм Шора и задача о дискретном логарифме

Задача о дискретном логарифме заключается в том, чтобы найти x, зная $y = g^x \bmod p$ для известных g и p. На классическом компьютере для ее решения необходимо экспоненциальное время, а алгоритм Шора – благодаря эффективному способу нахождения периода – позволяет найти x легко.

Например, рассмотрим функцию $\mathbf{f}(a, b) = g^a y^b$. Пусть требуется найти период этой функции, т. е. такие числа ω и ω', что $\mathbf{f}(a + \omega, b + \omega') = \mathbf{f}(a, b)$ для любых a и b. Тогда искомое решение $x = -\omega/\omega'$ по модулю q, где q – порядок g, т. е. известный параметр. Из равенства $\mathbf{f}(a + \omega, b + \omega') = \mathbf{f}(a, b)$ следует, что $g^{\omega} y^{\omega'} \bmod p = 1$. Подставляя g^x вместо y, получаем $g^{\omega + x\omega'} \bmod p = 1$, что эквивалентно $\omega + x\omega' \bmod q = 0$, откуда находим $x = -\omega/\omega'$.

Как и раньше, полная сложность равна $O(n^2(\log n)(\log\log n))$, где n – длина p в битах. Этот алгоритм обобщается на нахождение дискретных логарифмов в любой коммутативной группе, а не только в группе чисел по простому модулю.

Алгоритм Гровера

После алгоритма Шора, который экспоненциально ускоряет факторизацию, другой важной формой квантового ускорения является поиск в множестве n элементов за время, пропорциональное квадратному корню из n (напомним, что любой классический алгоритм требует времени, пропорционального n). Такое квадратичное ускорение возможно благодаря квантовому *алгоритму Гровера*, открытому в 1996 году (вскоре после алгоритма Шора). Я не стану распространяться о внутреннем механизме алгоритма Гровера, который, по существу, состоит из нескольких вентилей Адамара, но объясню, какую задачу он решает и каково его потенциальное воздействие на криптографическую безопасность. А также покажу, почему симметричный криптографический алгоритм можно спасти от квантовых компьютеров, удвоив размер ключа или хеш-значения, тогда как асимметричные алгоритмы обречены.

Алгоритм Гровера можно рассматривать как способ поиска среди n элементов такого значения x, для которого $\mathbf{f}(x) = 1$, притом что для большинства других элементов $\mathbf{f}(x) = 0$. Если m значений x удовлетворяют условию $\mathbf{f}(x) = 1$, то алгоритм Гровера найдет решение за время $O(\sqrt{n/m})$, т. е. пропорциональное квадратному корню из частного от деления n на m. Для сравнения: классический алгоритм справляется с этой задачей за время $O(n/m)$.

Теперь примем во внимание, что $\mathbf{f}()$ может быть произвольной функцией. Например, «$\mathbf{f}(x) = 1$ тогда и только тогда, когда x равно неизвестному секретному ключу K такому, что $\mathbf{E}(K, P) = C$» для некото-

рого известного открытого текста P и шифртекста C, где **E**() – какая-то функция шифрования. На практике это означает, что 128-битовый ключ AES можно найти на квантовом компьютере за время, пропорциональное 2^{64}, а не 2^{128}, как на классическом компьютере. Потребуется достаточно большой открытый текст, чтобы гарантировать уникальность ключа. (Если длина открытого и шифртекста равна, скажем, 32 бита, то найдется много ключей, отображающих открытый текст в шифртекст.) Сложность 2^{64} гораздо меньше, чем 2^{128}, а значит, добыть секретный ключ будет намного проще. Но и решение проблемы простое: чтобы восстановить 128-битовую безопасность, нужно просто использовать 256-битовые ключи! Тогда алгоритм Гровера сведет сложность поиска ключа «всего» к $2^{256/2} = 2^{128}$ операциям.

Алгоритм Гровера может также находить прообразы функций хеширования (см. главу 6). Чтобы найти прообраз некоторого значения h, определим функцию **f**() следующим образом: «**f**(x) = 1 тогда и только тогда, когда **Hash**(x) = h, в противном случае **f**(x) = 0». Таким образом, для нахождения прообраза n-битового хеш-значения потребуется порядка $2^{n/2}$ операций. Как и в случае шифрования, чтобы гарантировать *постквантовую* безопасность 2^n, достаточно вдвое увеличить длину хеш-значения.

Примечание *Существует квантовый алгоритм, который находит коллизии хеш-функций за время $O(2^{n/3})$, а не $O(2^{n/2})$, как в случае классической атаки на основе парадокса дней рождения. Можно было бы предположить, что квантовые компьютеры превосходят классические и для этой задачи, но беда в том, что этот квантовый алгоритм также требует памяти порядка $O(2^{n/3})$. Дайте столько памяти классическому алгоритму, и он сможет выполнить параллельный поиск коллизий за время всего $O(2^{n/6})$, что намного лучше, чем $O(2^{n/3})$. (Подробное описание этой атаки см. в статье Daniel J. Bernstein «Cost Analysis of Hash Collisions» по адресу http://cr.yp.to/papers.html#collisioncost.)*

Почему так трудно построить квантовый компьютер?

Хотя в принципе квантовые компьютеры создать можно, мы не знаем, насколько это будет трудно и когда случится, да и случится ли вообще. Пока что задача выглядит очень сложной. В начале 2017 года рекорд принадлежал машине, способной поддерживать в стабильном состоянии всего 14 кубитов на протяжении нескольких миллисекунд, тогда как для взлома любого криптографического алгоритма требуется поддерживать в стабильном состоянии миллионы кубитов на протяжении нескольких недель. Так что есть к чему стремиться.

Почему же так трудно построить квантовый компьютер? Потому что роль кубитов могут играть чрезвычайно мелкие объекты – размером с электрон или фотон. А будучи настолько мелкими, кубиты оказываются чрезвычайно хрупкими.

Кроме того, чтобы сохранять стабильность, кубиты должны работать при очень низких температурах (близких к абсолютному нулю). Но даже при таких температурах состояние кубитов деградирует, и в конечном итоге они становятся бесполезными. На момент написания этой книги неизвестно, как изготовить кубиты, которые жили бы больше пары секунд.

Еще одна проблема заключается в том, что кубиты подвержены влиянию окружающей среды, например тепла или магнитных полей. Создаваемый шум может приводить к ошибкам вычислений. Теоретически с такими ошибками можно справиться (если их частота не слишком велика), но это трудно. Для исправления ошибок кубитов нужны специальные методы – квантовые коды с исправлением ошибок, – которые, в свою очередь, требуют дополнительных кубитов и достаточно низкой частоты ошибок. Но как построить системы с такой низкой частотой ошибок, мы не знаем.

В настоящее время существует два основных подхода к формированию кубитов и, следовательно, к построению квантовых компьютеров: сверхпроводящие цепи и ионные ловушки. Идея *сверхпроводящих цепей* разрабатывается в лабораториях Google и IBM. Она основана на формировании кубитов в виде крохотных электрических цепей, опирающихся на квантовые явления в сверхпроводниках; носителями заряда в них являются пары электронов. Для кубитов на сверхпроводящих цепях должна поддерживаться температура, близкая к абсолютному нулю, а время их жизни очень мало. На момент написания книги рекордом были девять кубитов, сохраняющих стабильность в течение нескольких микросекунд.

Ионные ловушки, или захваченные ионы, состоят из ионов (заряженных атомов); с помощью лазера такие кубиты приводят в определенное начальное состояние. Использование ионных ловушек было одним из первых подходов к построению кубитов, в целом они демонстрируют большую стабильность, чем сверхпроводящие цепи. Рекорд составляет 14 кубитов, сохранявших стабильность в течение нескольких миллисекунд. Но ионные ловушки работают медленнее, а масштабировать их, похоже, труднее, чем сверхпроводящие цепи.

Построение квантового компьютера – поистине амбициозная задача. Проблема состоит из двух частей: 1) построить систему из небольшого числа кубитов, которая была бы стабильной, отказоустойчивой и позволяющей применить простые квантовые вентили, и 2) масштабировать такую систему на тысячи или миллионы кубитов, так чтобы она была практически полезной. С точки зрения чистой физики и при нынешнем состоянии наших знаний ничто не препятствует созданию больших отказоустойчивых квантовых компьютеров. Но многие вещи, возможные в теории, оказывалось трудно или слишком дорого реализовать на практике (например, безопасные компьютеры). Конечно, будущее покажет, кто прав – квантовые оптимисты (предсказывающие появление большого квантового компьютера в ближайшие десять лет) или квантовые скептики (доказывающие, что человечество никогда не увидит квантового компьютера).

Постквантовые криптографические алгоритмы

Предметом *постквантовой криптографии* является проектирование алгоритмов с открытым ключом, которые невозможно взломать на квантовом компьютере и которые, следовательно, смогут заменить RSA и алгоритмы на эллиптических кривых в будущем, когда стандартный квантовый компьютер будет щелкать 4096-битовые шифры RSA как орешки.

Такие алгоритмы не должны зависеть от трудной задачи, эффективно разрешимой с помощью алгоритма Шора, который легко справляется с трудностью задач факторизации и дискретного логарифмирования. Симметричные алгоритмы типа блочных шифров и функций хеширования хотя и потеряют половину своей теоретической безопасности при столкновении с квантовым компьютером, но пострадают не так сильно, как RSA. Они могли бы составить основу постквантовой схемы.

В следующих разделах я расскажу о четырех основных типах постквантовых алгоритмов: на основе кодов, на основе решеток, на основе многомерных систем и на основе функций хеширования. Из них мне больше всего нравятся алгоритмы на основе функций хеширования своей простотой и гарантиями стойкости.

Криптография на основе кодов

Постквантовые криптографические алгоритмы на основе кодов опираются на *коды, исправляющие ошибки*, – метод, предназначенный для передачи битов по зашумленному каналу. Основы теории кодов, исправляющих ошибки, были заложены в 1950-е годы. Первая схема шифрования на основе таких кодов (криптосистема *McEliece*) была разработана в 1978 году и до сих пор не взломана. Криптосхемы на основе кодов можно использовать как для шифрования, так и для цифровых подписей. Их главное ограничение – размер открытого ключа, обычно порядка нескольких сотен килобайт. Но разве это проблема, когда средний размер веб-страницы равен 2 мегабайтам?

Сначала разберемся, что такое коды, исправляющие ошибки. Пусть требуется передать последовательность битов в виде (например) последовательности 3-битовых слов, но канал ненадежен и есть опасность, что 1 или несколько битов будут переданы неправильно: вы передали 010, а получатель увидел 011. Простой способ решить эту проблему заключается в использовании кода, исправляющего ошибки: вместо 010 нужно передать 000111000 (каждый бит повторяется три раза). Тогда получатель сможет декодировать слово, выбрав из каждой группы трех битов тот, который встречается чаще. Например, 100110111 будет декодировано как 011. Но легко видеть, что этот код позволяет получателю исправить не более одной ошибки в группе из трех битов, потому что если в одной группе встретилось две ошибки, то чаще будет встречаться не правильный, а как раз ошибочный бит.

Линейные коды – пример не столь тривиального кода, исправляющего ошибки. В этом случае кодируемое слово рассматривается как n-битовый вектор v, а кодирование заключается в умножении v на матрицу G размера $m \times n$, в результате чего получается кодовое слово $w = vG$. (В этом примере m больше n, т. е. кодовое слово длиннее исходного.) Матрицу можно построить так, что для заданного числа t любая ошибка в t битах слова w может быть исправлена получателем, который получит правильное v. Иными словами, t – максимальное количество допускающих исправление ошибок.

Чтобы зашифровать данные с помощью линейных кодов, в криптосистеме McEliece G строится как секретная комбинация трех матриц, а шифрование сводится к вычислению $w = vG$ плюс некоторое случайное значение e, содержащее фиксированное число единичных битов. Здесь G – открытый ключ, а закрытый ключ состоит из матриц A, B и C таких, что $G = ABC$. Зная A, B и C, можно надежно декодировать сообщение и получить w. (Описание шага декодирования можете найти в сети.)

Безопасность схемы шифрования McEliece опирается на трудность декодирования линейного кода при недостаточной информации. Известно, что это **NP**-полная задача, которая, следовательно, не по зубам квантовым компьютерам.

Криптография на основе решеток

Решетками называются математические образования, состоящие из множества точек в n-мерном пространстве и имеющие периодическую структуру. Например, в двумерном случае ($n = 2$) решетку можно рассматривать как множество точек, показанное на рис. 14.5.

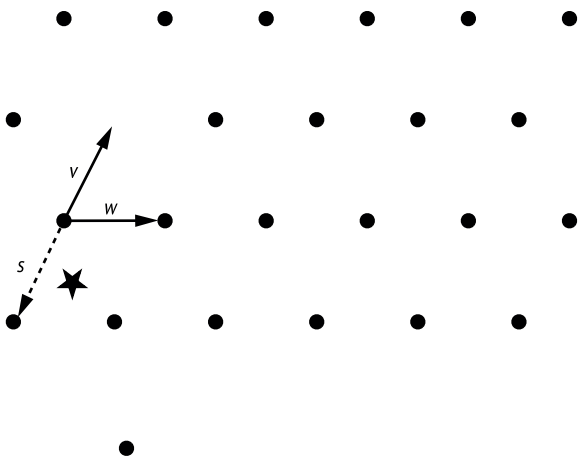

Рис. 14.5. Точки двумерной решетки, где v и w – базисные векторы решетки, а s – вектор, ближайший к точке, обозначенной звездочкой

Теория решеток породила обманчиво простые криптографические схемы. Я опишу, в чем их суть.

Первая трудная задача в криптографии на основе решеток известна под названием *короткое целочисленное решение* (short integer solution – SIS). Она заключается в том, чтобы найти секретный вектор s, содержащий n чисел, если известны (A, b) такие, что $b = As \bmod q$, где A – случайная матрица размера $m \times n$, а q – простое число.

Вторая трудная задача называется *обучением с ошибками* (learning with errors – LWE) и заключается в нахождении секретного вектора s, содержащего n чисел, если известны (A, b), где $b = As + e \bmod q$, A – случайная матрица размера $m \times n$, e – случайный вектор шума, а q – простое число. Эта задача очень похожа на задачу декодирования при зашумленных каналах в криптографии на основе кодов.

Задачи SIS и LWE эквивалентны и могут быть переформулированы как *задача о ближайшем векторе* (closest vector problem – CVP) на решетке, которая заключается в нахождении ближайшего к данной точке вектора решетки, представленного в виде линейной комбинации базисных векторов. Пунктирный вектор s на рис. 14.5 показывает, как можно было бы найти ближайший к обозначенной звездочкой точке вектор, равный линейной комбинации базисных векторов v и w.

CVP и другие задачи на решетках считаются трудными для классических и квантовых компьютеров. Но это не значит, что они так просто порождают безопасные криптосистемы, потому что некоторые задачи трудны только в худшем случае (т. е. их самые трудные экземпляры), а не в среднем (а для криптографии необходимо именно это свойство). Кроме того, даже когда найти точное решение CVP трудно, нахождение приближения к нему может оказаться значительно более легкой задачей.

Криптография на основе многомерных систем

Криптография на основе многомерных систем – это построение криптографических схем, взломать которые так же трудно, как решить многомерную систему уравнений, т. е. систему уравнений со многими неизвестными. Рассмотрим, к примеру, следующую систему уравнений с четырьмя неизвестными x_1, x_2, x_3, x:

$$
\begin{aligned}
&x_1x_2 + x_3x_4 + x_2 = 1;\\
&x_1x_3 + x_1x_4 + x_2x_3 = 12;\\
&x_1^2 + x_3^2 + x_4 = 4;\\
&x_2x_3 + x_2x_4 + x_1 + x_4 = 0.
\end{aligned}
$$

Каждое уравнение включает сумму членов, содержащих либо одно неизвестное, например x_4 (члены первой степени), либо произведение двух неизвестных, например x_2x_3 (*квадратичные* члены). Чтобы решить эту систему, нужно найти значения x_1, x_2, x_3, x_4, которые удовлетворяют всем четырем уравнениям. Уравнения могут рассматриваться над множеством всех вещественных чисел, только над целыми

числами или над конечными множествами чисел. В криптографии обычно рассматриваются уравнения над полем чисел по простому модулю или над двоичными значениями (0 и 1).

Задача нахождения решения является **NP**-трудной для *случайной* квадратичной системы уравнений. Эта трудная задача о *многомерных системах квадратичных уравнений* (multivariate quadratics – MQ) может стать основой для постквантовых систем, потому что квантовые компьютеры не умеют эффективно решать **NP**-трудные задачи.

К сожалению, построить криптосистему на основе MQ не так-то просто. Например, в случае применения MQ для цифровых подписей закрытый ключ мог бы состоять из трех систем уравнений, L_1, N и L_2, которые при объединении в этом порядке давали бы еще одну систему уравнений, P, являющуюся открытым ключом. Тогда последовательное применение преобразований L_1, N и L_2 (т. е. преобразование группы значений, как предписано системой уравнений) эквивалентно применению P путем преобразования x_1, x_2, x_3, x_4 в y_1, y_2, y, y_4 следующим образом:

$$y_1 = x_1x_2 + x_3x_4 + x_2;$$
$$y_2 = x_1x_3 + x_1x_4 + x_2x_3;$$
$$y_3 = x_1^2 + x_3^2 + x_4;$$
$$y_4 = x_2x_3 + x_2x_4 + x_1 + x_4.$$

В такой криптосистеме L_1, N и L_2 выбираются так, что L_1 и L_2 – линейные преобразования (т. е. в составляющих уравнениях есть только сложения, а умножений нет), причем обратимые, а N – квадратичная система уравнений, тоже обратимая. Поэтому комбинация всех трех систем является обратимой квадратичной системой, но обратить ее трудно, не зная обращений L_1, N и L_2.

Вычисление подписи тогда сводится к вычислению обращений L_1, N и L_2 в применении к некоторому сообщению M, рассматриваемому как последовательность переменных $x_1, x_2, \ldots$:

$$S = L_2^{-1}(N^{-1}(L_1^{-1}(M))).$$

Верификация подписи заключается в проверке того, что $P(S) = M$.

Противник смог бы взломать такую криптосистему, если бы сумел вычислить обращение P или определить L_1, N и L_2, зная P. Трудность подобных задач зависит от параметров схемы, например количества уравнений, размера и типа чисел и т. д. Но выбрать безопасные параметры нелегко, поэтому не одна схема на основе многомерных систем, считавшаяся безопасной, была взломана.

Криптография на основе многомерных систем не используется в серьезных приложениях из-за опасений по поводу ее безопасности, а также потому, что работает медленно и требует очень много памяти. Но у схем подписания на основе многомерных систем есть одно достоинство – они порождают короткие подписи.

Криптография на основе функций хеширования

В отличие от предыдущих схем, криптография на основе функций хеширования базируется на хорошо изученной безопасности криптографических хеш-функций, а не на трудности математических задач. Поскольку квантовые компьютеры не могут взломать хеш-функции, они не смогут взломать никакую схему, основанную на трудности нахождения коллизий, а именно это ключевая идея схем цифровой подписи на базе хеш-функций.

Криптографические схемы на основе функций хеширования довольно сложны, поэтому я остановлюсь только на самом простом из входящих в них строительных блоков: одноразовой подписи – приеме, открытом примерно в 1979 году и получившем название *одноразовая подпись Винтерница* (Winternitz one-time signature – WOTS) по имени автора. Здесь «одноразовая» означает, что закрытый ключ можно использовать для подписания только одного сообщения, иначе схема станет небезопасной. (WOTS можно сочетать с другими методами для подписания нескольких сообщений, как мы увидим в следующем разделе.)

Но сначала посмотрим, как работает WOTS. Пусть требуется подписать сообщение, рассматриваемое как число от 0 до $w - 1$, где w – параметр схемы. Закрытым ключом является случайная строка K. Чтобы подписать сообщение M, где $0 \leqslant M < w$, мы вычисляем **Hash**(**Hash**(…(**Hash**(K))), где хеш-функция **Hash** повторяется M раз. Обозначим это значение $\mathbf{Hash}^M(K)$. Открытым ключом является $\mathbf{Hash}^w(K)$, или результат w вложенных итераций **Hash**, начиная с K.

WOTS-подпись, S, верифицируется путем проверки того, что $\mathbf{Hash}^{w-M}(S)$ равно открытому ключу $\mathbf{Hash}^w(K)$. Заметим, что S – это K после M применений **Hash**, поэтому если еще $w - M$ раз применить **Hash**, то мы получим значение, равное K, хешированному $M + (w - M) = w$ раз, а это и есть открытый ключ.

Эта схема выглядит довольно непритязательно и имеет ряд существенных ограничений.

Подписи можно подделать

Зная $\mathbf{Hash}^M(K)$, подпись M, можно вычислить $\mathbf{Hash}(\mathbf{Hash}^M(K)) = \mathbf{Hash}^{M+1}(K)$, т. е. правильную подпись сообщения $M + 1$. Эту проблему можно решить, если подписывать не только M, но и $w - M$, используя второй ключ.

Схема работает только для коротких сообщений

Если сообщения 8-битовые, то всего их существует $2^8 - 1 = 255$, так что для создания подписи нужно будет вычислить **Hash** 255 раз. Для коротких сообщений это может работать, а для длинных – нет: например, подписание 128-битовых сообщений потребовало бы $2^{128} - 1$ вычислений **Hash** и продолжалось бы вечно. Решение – разбивать длинные сообщения на несколько коротких.

Схема работает только один раз

Если закрытый ключ используется для подписания более одного сообщения, то противник может восстановить достаточно информации для подделки подписи. Например, если $w = 8$ и мы подписали числа 1 и 7, воспользовавшись описанным выше трюком, чтобы избежать тривиальных подделок, то противник получает $\mathbf{Hash}^1(K)$ и $\mathbf{Hash}^7(K')$ в качестве подписи 1 и $\mathbf{Hash}^7(K)$ и $\mathbf{Hash}^1(K')$ в качестве подписи 7. Зная эти значения, противник может вычислить $\mathbf{Hash}^x(K)$ и $\mathbf{Hash}^x(K')$ для любого x из [1;7] и, следовательно, подделать подпись владельца K и K'. И нет никакого простого способа это исправить.

Современные схемы на основе функций хеширования опираются на более сложные варианты WOTS в сочетании с древовидными структурами данных и изощренными методами, придуманными для подписания разных сообщений разными ключами. К сожалению, получающиеся схемы порождают длинные подписи (порядка десятков килобайт, как в случае SPHINCS, одной из самых передовых схем на момент написания этой книги).

Какие возможны проблемы

Постквантовая криптография может оказаться принципиально более стойкой, чем RSA или эллиптическая криптография, но и она не может считаться непогрешимой или всемогущей. Наше понимание безопасности постквантовых схем и их реализаций отстает от понимания традиционной криптографии, и это несет дополнительные риски, кратко описанные в следующих разделах.

Непонятный уровень безопасности

Постквантовые схемы могут показаться обманчиво стойкими, хотя на деле уязвимы как к квантовым, так и к классическим атакам. Алгоритмы на основе решеток, в частности семейство вычислительных задач на базе обучения с ошибками в кольце (варианты задачи LWE, работающие с полиномами), иногда оказываются проблематичными. LWE в кольце привлекает криптографов тем, что на его базе можно построить криптосистемы, которые в принципе так же трудно взломать, как решить самые трудные задачи LWE в кольце, которые могут быть **NP**-трудными. Но если безопасность выглядит слишком хорошо, чтобы быть правдой, зачастую стоит усомниться.

Проблема с доказательствами безопасности заключается в том, что они часто асимптотические, т. е. верны только при большом числе параметров, например при большой размерности решетки. Однако на практике параметров используется гораздо меньше.

Даже если кажется, что схему на основе решеток взломать так же трудно, как решить некоторую **NP**-трудную задачу, количественно

оценить ее безопасность все равно трудно. Для таких алгоритмов мы редко имеем ясную картину возможных атак и их стоимости в терминах объема вычислений или оборудования, поскольку еще плохо понимаем эти не так давно предложенные конструкции. Из-за этой неопределенности схемы на основе решеток трудно сравнивать с хорошо изученными конструкциями типа RSA, что отпугивает потенциальных пользователей. Однако на этом фронте наблюдается прогресс и есть надежда, что через несколько лет задачи на решетках будут так же понятны, как RSA. (Технические детали, касающиеся задачи LWE в кольце, см. в отличном обзоре Пейкерта по адресу *https://eprint.iacr.org/2016/351/*.)

Забегая вперед: что, если уже слишком поздно?

Представьте себя кричащий заголовок CNN 1 апреля 2048 года: «ACME, Inc. объявляет о своем созданном в секрете квантовом компьютере и запускает платформу "взлом криптосистем как услуга"». RSA и эллиптическая криптография больше не стоят ни гроша. И что дальше?

Важно, что постквантовое шифрование более критично, чем постквантовые подписи. Сначала рассмотрим случай подписей. Если бы вы к тому времени еще пользовались схемой подписания на основе RSA-PSS или ECDSA, то могли бы просто выпустить новые подписи с использованием постквантовой схемы и таким образом восстановить доверие к своим подписям. После этого вы могли бы отозвать старые квантовонебезопасные открытые ключи и вычислить свежие подписи для каждого подписанного вами сообщения. Конечно, придется поработать, но в конечном итоге все будет хорошо.

Впадать в панику стоит, только если вы шифровали данные с помощью квантовонебезопасных схем типа RSA-OAEP. В таком случае все переданные ранее шифртексты могут быть скомпрометированы. Очевидно, что нет никакого смысла заново зашифровывать открытые тексты постквантовым алгоритмом, потому что конфиденциальность данных уже утрачена.

А как насчет протоколов совместной выработки ключа, например Диффи–Хеллмана (DH) и его эллиптического аналога (ECDH)?

На первый взгляд, ситуация выглядит такой же плохой, как для шифрования: противник, знающий открытые ключи g^a и g^b, мог бы воспользоваться своим новеньким квантовым компьютером для вычисления секретного показателя степени a или b, вычислить разделяемый секрет g^{ab}, а затем сформировать на его основе ключи для дешифрирования вашего трафика. Но на практике протокол Диффи–Хеллмана не всегда используется так примитивно. Для формирования настоящих сеансовых ключей, применяемых для шифрования данных, нужно знать как разделяемый секрет Диффи–Хеллмана, так и некоторое внутреннее состояние вашей системы.

Вот, например, как работают современные мобильные системы обмена сообщениями благодаря протоколу, впервые реализованному

в приложении Signal. Когда вы посылаете новое сообщение собеседнику в Signal, вычисляется новый разделяемый секрет Диффи–Хеллмана, который объединяется с внутренними секретами, зависящими от предыдущих сообщений, отправленных в данном сеансе (а он может продолжаться довольно долго). Такое творческое применение протокола Диффи–Хеллмана сильно затрудняет атаку, даже на квантовом компьютере.

Проблемы реализации

На практике постквантовые схемы будут представлять собой код, а не алгоритмы, т. е. программное обеспечение, работающее на некотором физическом процессоре. И каким бы стойким ни был алгоритм на бумаге, он не будет защищен от ошибок реализации, программных дефектов и атак по побочным каналам. Алгоритм может быть от начала до конца постквантовым в теории и все же допускать взлом на классическом компьютере, потому что программист забыл ввести точку с запятой.

Кроме того, схемы, основанные на кодах, исправляющих ошибки, и на решетках, сильно зависят от математических операций, при реализации которых нужно применять разнообразные приемы, чтобы добиться максимального быстродействия. Но тогда код становится чрезмерно сложным, что может сделать реализацию более уязвимой к атакам по побочному каналу, например с хронометражем, в ходе которых противник добывает информацию о секретных значениях, измеряя время выполнения. На самом деле такие атаки уже применялись против шифрования на основе кодов (см. *https://eprint.iacr.org/2010/479/*) и против схем подписания на основе решеток (см. *https://eprint.iacr.org/2016/300/*).

Таким образом, как это ни парадоксально, постквантовые схемы поначалу будут менее безопасны на практике, чем классические, – из-за уязвимостей в реализации.

Для дополнительного чтения

Желающим почитать об основах квантовых вычислений рекомендую классическую книгу Nielsen and Chuang «Quantum Computation and Quantum Information» (Cambridge, 2000)[1]. Книга Aaronson «Quantum Computing Since Democritus» (Cambridge, 2013), менее техническая и написанная более живо, рассказывает не только о квантовых вычислениях.

Существует несколько программных эмуляторов, позволяющих экспериментировать с квантовыми вычислениями. Особенно хорошо спроектирован сайт Quantum Computing Playground (*http://www.quan-*

[1] *Нильсен М., Чанг И.* Квантовые вычисления и квантовая информация. М.: Мир, 2006.

tumplayground.net/), предлагающий простой язык программирования и интуитивно понятные визуализации.

О последних исследованиях по постквантовой криптографии можно узнать на сайте *https://pqcrypto.org/* и связанной с ними конференции PQCrypto.

Будущее обещает много интересного для постквантовой криптографии благодаря проекту NIST Post-Quantum Crypto, в рамках которого сообщество старается разработать будущий постквантовый стандарт. Обязательно загляните на сайт проекта *http://csrc.nist.gov/groups/ST/post-quantum-crypto/*, где выкладываются относящиеся к теме алгоритмы, научные статьи и семинары.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Жан-Филипп Омассон

О криптографии всерьез

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 26,65. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

«Подробное рассмотрение современной криптографической практики, которое позволит специалистам в области шифрования улучшить свои навыки».
Мэттью Д. Грин,
профессор Института информационной безопасности университета Джонса Хопкинса

В данном практическом руководстве по современному шифрованию анализируются фундаментальные математические идеи, лежащие в основе криптографии. Рассказывается о шифровании с аутентификацией, безопасной случайности, функциях хеширования, блочных шифрах и методах криптографии с открытым ключом, в частности RSA и криптографии на эллиптических кривых.

Каждая глава содержит обсуждение типичных ошибок реализации с примерами из практики и подробное описание возможных проблем, сопровождаемое рекомендациями по их устранению.

Вы узнаете:

- *о ключевых понятиях криптографии: вычислительной безопасности, моделях атак и секретности прошлого;*
- *о стойкости и ограничениях протокола TLS, лежащего в основе безопасных веб-сайтов, работающих по протоколу HTTPS;*
- *о квантовых вычислениях и постквантовой криптографии;*
- *о различных уязвимостях, иллюстрируемых примерами кода и сценариями использования;*
- *о том, как выбирать наилучший алгоритм или протокол и как задавать правильные вопросы поставщику.*

Независимо от того, занимаетесь вы разработкой профессионально или только начинаете знакомство с предметом, в этой книге вы найдете полный обзор современной криптографии и ее приложений.

Жан-Филипп Омассон – *главный инженер-исследователь в международной компании Kudelski Security со штаб-квартирой в Швейцарии, занимающейся кибербезопасностью. Он автор более 40 научных статей по криптографии и криптоанализу. Спроектировал широко используемые хеш-функции BLAKE2 и SipHash. Регулярно выступает на конференциях по безопасности, проводил презентации на Black Hat, DEF CON, Troopers и Infiltrate.*

ISBN 978-5-97060-975-0

Интернет-магазин:
www.dmkpress.com
Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф