Серьёзная криптография. Практическое введение в современное шифрование

Русский перевод практического руководства Жан-Филиппа Омассона по современным криптографическим алгоритмам, протоколам, инженерным решениям и типичным ошибкам реализации.

Больше крутых материалов в моём телеграм-канале @grogrigs

Глава 1. Шифрование

Шифрование - основная область применения криптографии; оно делает данные непонятными, обеспечивая их конфиденциальность. Для шифрования используются алгоритм, называемый шифром, и секретное значение, называемое ключом. Если вы не знаете секретного ключа, то не сможете ни расшифровать сообщение, ни получить хотя бы один бит информации о нём; не сможет этого и ни один злоумышленник.

Эта глава посвящена симметричному шифрованию - простейшему виду шифрования. При симметричном шифровании ключ расшифрования совпадает с ключом шифрования (в отличие от асимметричного шифрования, или шифрования с открытым ключом, где ключи различаются). Сначала вы познакомитесь с самыми слабыми формами симметричного шифрования - классическими шифрами, способными противостоять лишь совершенно неграмотному противнику, а затем мы перейдём к самым сильным формам, которые останутся безопасными навсегда.

Основы

При шифровании сообщения открытым текстом называют незашифрованное сообщение, а шифротекстом - зашифрованное. Следовательно, шифр состоит из двух функций: функция шифрования преобразует открытый текст в шифротекст, а функция расшифрования вновь превращает шифротекст в открытый текст. Однако мы нередко будем говорить «шифр», подразумевая именно «шифрование». Например, на рис. 1-1 показан шифр `E` в виде блока, который принимает на вход открытый текст `P` и ключ `K`, а на выходе выдаёт шифротекст `C`. Это отношение я буду записывать как `C = E(K, P)`. Аналогично, когда шифр работает в режиме расшифрования, я буду писать `D(K, C)`.

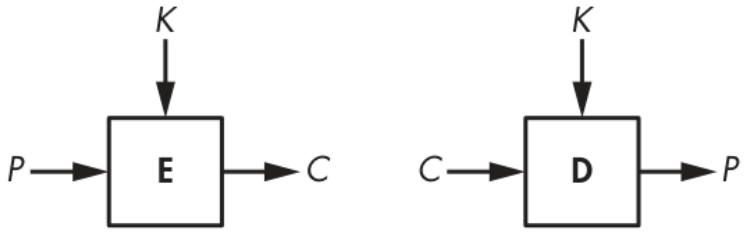

Рисунок 1-1. Базовые операции шифрования и расшифрования

> **Примечание.** У некоторых шифров шифротекст имеет тот же размер, что и открытый текст; у других шифротекст немного длиннее. Однако шифротекст никогда не может быть короче открытого текста.

Классические шифры

Классические шифры появились раньше компьютеров и потому работают с буквами, а не с битами, что делает их значительно проще современного шифра вроде Data Encryption Standard. Например, в Древнем Риме или во время Первой мировой войны нельзя было воспользоваться вычислительной мощностью компьютерной микросхемы, чтобы перемешать сообщение: всё приходилось делать лишь ручкой и бумагой. Классических шифров существует много, но самые известные из них - шифр Цезаря и шифр Виженера.

Шифр Цезаря

Шифр Цезаря получил своё название потому, что римский историк Светоний сообщал о его использовании Юлием Цезарем. Этот шифр шифрует сообщение, сдвигая каждую букву на три позиции вперёд по алфавиту и возвращаясь к A, если при сдвиге достигнута Z. Например, ZOO шифруется как CRR, FDHVDU расшифровывается как CAESAR и так далее, как показано на рис. 1-2. В числе 3 нет ничего особенного; просто такой сдвиг легче вычислить в уме, чем сдвиг на 11 или 23 позиции.

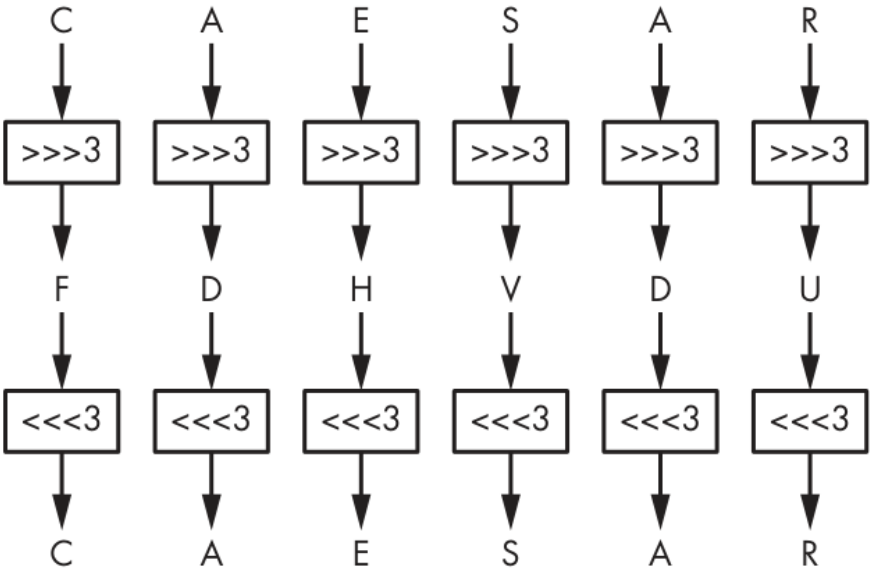

Рисунок 1-2. Шифр Цезаря

Шифр Цезаря чрезвычайно легко взломать: чтобы расшифровать данный шифротекст, достаточно сдвинуть буквы на три позиции назад и получить открытый текст. Тем не менее во времена Красса и Цицерона шифр Цезаря, возможно, был достаточно стойким. Поскольку секретного ключа в нём нет (сдвиг всегда равен 3), пользователи шифра Цезаря исходили из того, что злоумышленники неграмотны или недостаточно образованны, чтобы во всём разобраться, - сегодня такое предположение гораздо менее реалистично. (На самом деле в 2006 году итальянская полиция арестовала босса мафии после расшифрования записок на клочках бумаги, зашифрованных одним из вариантов шифра Цезаря: например, ABC в нём превращалось в 456, а не в DEF.)

Можно ли сделать шифр Цезаря более стойким? Можно представить его версию, в которой вместо постоянного значения 3 используется секретная величина сдвига, но это мало чем поможет: злоумышленник сможет перебрать все 25 возможных значений, пока расшифрованное сообщение

не приобретёт смысл.

Шифр Виженера

Потребовалось около 1500 лет, чтобы шифр Цезаря получил существенное усовершенствование в виде шифра Виженера, созданного в XVI веке итальянцем Джованом Баттистой Беллазо. Название «Виженер» происходит от имени француза Блеза де Виженера, который в XVI веке изобрёл другой шифр, однако из-за исторически сложившейся ошибочной атрибуции за этим шифром закрепилось имя Виженера. Как бы то ни было, шифр Виженера приобрёл популярность и позднее применялся, помимо прочих, войсками Конфедерации во время Гражданской войны в США и швейцарской армией во время Первой мировой войны.

Шифр Виженера похож на шифр Цезаря, но буквы в нём сдвигаются не на три позиции, а на значения, заданные ключом - набором букв, представляющих числа в соответствии с положением букв в алфавите. Например, если ключ равен DUH, буквы открытого текста сдвигаются на 3, 20 и 7 позиций, поскольку D находится на три буквы дальше A, U - на 20 букв дальше A, а H - на семь букв дальше A. Последовательность 3, 20, 7 повторяется, пока не будет зашифрован весь открытый текст. Например, при использовании ключа DUH слово CRYPTO будет зашифровано как FLFSNV: C сдвигается на три позиции и превращается в F, R сдвигается на 20 позиций и превращается в L и так далее. На рис. 1-3 этот принцип показан при шифровании предложения THEY DRINK THE TEA.

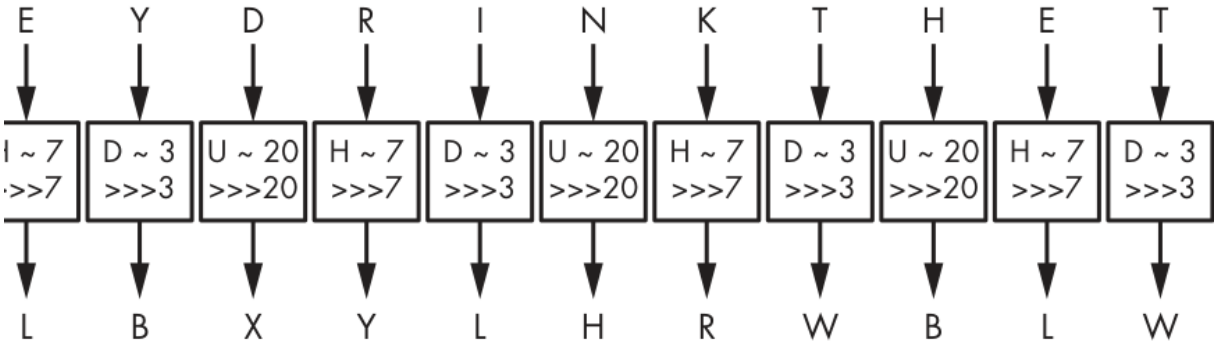

Рисунок 1-3. Шифр Виженера

Шифр Виженера явно надёжнее шифра Цезаря, однако взломать его всё ещё довольно легко. Первый шаг при расшифровании - определить длину ключа. Возьмём пример с рис. 1-3, где THEY DRINK THE TEA шифруется как WBLBXYLHRWBLWYH с помощью ключа DUH. (Пробелы обычно удаляют, чтобы скрыть границы слов.) Обратите внимание: в шифротексте WBLBXYLHRWBLWYH трёхбуквенная группа WBL встречается дважды с интервалом в девять букв. Это наводит на мысль, что одно и то же трёхбуквенное слово было зашифровано с одинаковыми величинами сдвига и оба раза дало WBL. После этого криптоаналитик может заключить, что длина ключа равна либо девяти, либо делителю девяти (то есть трём). Более того, он может предположить, что повторяющееся трёхбуквенное слово - это THE, и тем самым определить DUH как возможный ключ шифрования.

Второй шаг при взломе шифра Виженера - определить сам ключ методом частотного анализа, который использует неравномерность распределения букв в языках. Например, в английском языке E - самая распространённая буква, поэтому, если наиболее частой буквой шифротекста оказалась X, самым вероятным значением открытого текста в этой позиции будет E.

Несмотря на относительную слабость, в своё время шифр Виженера мог обеспечивать достаточно надёжное шифрование сообщений. Возможности частотного анализа ограничены: для него требуется несколько предложений, а значит, он не сработает, если шифром зашифрованы

короткие сообщения. Кроме того, большинству сообщений требовалось оставаться секретными лишь непродолжительное время, поэтому не имело значения, если противник в конце концов расшифровывал шифротекст. (Криптограф XIX века Огюст Керкгоффс оценивал необходимый срок конфиденциальности большинства зашифрованных сообщений военного времени всего в три-четыре часа.)

Как работают шифры

Опираясь на простейшие шифры Цезаря и Виженера, мы можем попытаться абстрактно описать работу шифра, выделив два его главных компонента: перестановку и режим работы. Перестановка - это функция, преобразующая элемент (в криптографии - букву или группу битов) так, что для каждого элемента существует единственный обратный элемент (примером служит сдвиг на три буквы в шифре Цезаря).

Режим работы - это алгоритм, использующий перестановку для обработки сообщений произвольного размера. Режим шифра Цезаря тривиален: для каждой буквы просто повторяется одна и та же перестановка. Но, как вы уже видели, у шифра Виженера режим сложнее: к буквам в разных позициях применяются разные перестановки.

В следующих разделах я подробнее расскажу, что представляют собой эти компоненты и как они связаны со стойкостью шифра. На примере каждого из них я покажу, почему классические шифры обречены быть небезопасными в отличие от современных шифров, выполняемых высокоскоростными компьютерами.

Перестановка

Большинство классических шифров заменяет каждую букву другой буквой - иными словами, выполняет подстановку. В шифрах Цезаря и Виженера подстановка представляет собой сдвиг в алфавите, хотя алфавит или набор символов может быть иным: вместо английского алфавита может использоваться арабский, а вместо букв, например, слова, числа или идеограммы. Представление или кодирование информации - отдельный вопрос, по большей части не имеющий отношения к безопасности. (Мы рассматриваем латинские буквы, поскольку именно их используют классические шифры.)

Подстановкой в шифре не может быть любая произвольная подстановка. Это должна быть перестановка - такое переупорядочение букв от A до Z, при котором у каждой буквы имеется единственный обратный образ. Например, подстановка, преобразующая буквы A, B, C и D соответственно в C, A, D и B, является перестановкой, поскольку каждой букве соответствует ровно одна другая буква. Но подстановка, преобразующая A, B, C, D в D, A, A, C, перестановкой не является, поскольку и B, и C отображаются в A. При перестановке у каждой буквы имеется ровно один обратный образ.

Однако не всякая перестановка безопасна. Чтобы обеспечивать безопасность, перестановка шифра должна удовлетворять трём критериям:

• **Перестановка должна определяться ключом.** Это сохраняет перестановку в тайне до тех пор, пока остаётся секретным ключ. Если в шифре Виженера вы не знаете ключ, то не знаете и того, какая из 26 перестановок применялась, а потому не можете с лёгкостью выполнить расшифрование.

• **Разные ключи должны приводить к разным перестановкам.** В противном случае расшифровать сообщение без ключа становится легче: если разные ключи дают одинаковые перестановки, значит, различных ключей меньше, чем различных перестановок, а следовательно, при расшифровании без ключа нужно перебрать меньше вариантов. В шифре Виженера каждая буква ключа определяет подстановку; существует 26 различных букв и столько же различных перестановок.

• **Перестановка, грубо говоря, должна выглядеть случайной.** После выполнения перестановки в шифротексте не должно оставаться закономерностей, поскольку они делают перестановку предсказуемой для злоумышленника, а значит, менее стойкой. Например, подстановка в шифре Виженера вполне предсказуема: если для некоторого смещения вы установили, что A шифруется как F, можно заключить, что величина сдвига равна 5, а также узнать, что B шифруется как G, C - как H и так далее. Но при случайно выбранной перестановке знание того, что A шифруется как F, сообщило бы вам лишь одно: B не шифруется как F.

Перестановку, удовлетворяющую этим критериям, мы будем называть безопасной перестановкой. Как вы увидите далее, безопасная перестановка необходима для построения безопасного шифра, но сама по себе недостаточна. Шифру также требуется режим работы, позволяющий обрабатывать сообщения любой длины.

Режим работы

Предположим, у нас есть безопасная перестановка, которая преобразует, например, A в X, B в M, а N в L. Тогда слово BANANA шифруется как MXLXLX, причём каждое вхождение A заменяется на X. Таким образом, применение одной и той же перестановки ко всем буквам открытого текста раскрывает все повторяющиеся буквы. Анализируя эти повторы, вы, возможно, не узнаете сообщение целиком, но всё же получите некоторую информацию о нём. В примере с BANANA ключ не нужен, чтобы догадаться: в трёх позициях с X в открытом тексте стоит одна и та же буква, а в двух позициях с L - другая общая буква. Если известно, что сообщение - название фрукта, можно определить, что это BANANA, а не CHERRY, LYCHEE или название другого фрукта из шести букв.

Режим работы (или просто режим) шифра снижает утечку информации о повторяющихся буквах открытого текста, применяя к ним разные перестановки. Режим шифра Виженера решает эту задачу лишь частично: если длина ключа равна `N` буквам, то для каждых `N` последовательных букв используются `N` разных перестановок. Но и это может оставлять в шифротексте закономерности, поскольку к каждой `N`-й букве сообщения применяется одна и та же перестановка. Именно поэтому частотный анализ позволяет взломать шифр Виженера.

Частотный анализ можно сделать бесполезным, если шифр Виженера будет шифровать только открытые тексты той же длины, что и ключ. Но даже тогда останется ещё одна проблема: многократное использование одного ключа раскрывает сходство между открытыми текстами. Например, при ключе KYN слова TIE и PIE шифруются соответственно как DGR и ZGR. Оба шифротекста оканчиваются одинаковыми двумя буквами GR, показывая, что последние две буквы обоих открытых текстов также совпадают. В безопасном шифре обнаружить такие закономерности должно быть невозможно.

Чтобы построить безопасный шифр, нужно сочетать безопасную перестановку с безопасным режимом. В идеальном случае такое сочетание не позволяет злоумышленникам узнать о сообщении ничего, кроме его длины.

Почему классические шифры небезопасны

Классические шифры обречены быть небезопасными, поскольку ограничены операциями, которые можно выполнить в уме или на листе бумаги. Им недостаёт вычислительной мощности компьютера, поэтому простые компьютерные программы легко их взламывают. Посмотрим, по какой фундаментальной причине эта простота делает их небезопасными в современном мире.

Вспомним, что для безопасности перестановка шифра должна выглядеть случайной. Конечно, лучший способ выглядеть случайной - действительно быть случайной, то есть выбирать каждую перестановку случайно из множества всех перестановок. А выбирать есть из чего. В случае английского алфавита из 26 букв существует приблизительно 2^88 перестановок:

```
26! = 403291461126605635584000000 ~= 2^88
```

Здесь восклицательный знак (!) обозначает факториал, который определяется следующим образом:

```
n! = n * (n - 1) * (n - 2) * ... * 3 * 2.
```

(Чтобы понять, откуда берётся это число, посчитайте перестановки как списки букв в изменённом порядке: для первой буквы имеется 26 вариантов, затем 25 вариантов для второй, 24 для третьей и так далее.) Это огромное число: оно имеет тот же порядок величины, что и число атомов в человеческом теле. Но классические шифры способны использовать лишь малую долю этих перестановок, а именно те, для которых требуются простые операции (например, сдвиги) и существует короткое описание (вроде короткого алгоритма или небольшой таблицы поиска). Проблема в том, что безопасная перестановка не может одновременно удовлетворять обоим этим ограничениям.

Безопасные перестановки можно получать с помощью простых операций: выбрать случайную перестановку, представить её таблицей из 25 букв (этого достаточно для представления перестановки 26 букв, поскольку 26-я определяется по отсутствующему значению) и применять, находя буквы в этой таблице. Но тогда описание уже не будет коротким. Например, для описания десяти разных перестановок потребуется 250 букв вместо всего десяти букв, используемых в шифре Виженера.

Можно также получать безопасные перестановки с коротким описанием. Вместо простого сдвига алфавита можно применять более сложные операции, например сложение и умножение. Именно так работают современные шифры: получив ключ, обычно длиной 128 или 256 бит, они выполняют сотни битовых операций, чтобы зашифровать одну букву. На компьютере, способном выполнять миллиарды битовых операций в секунду, этот процесс быстр, но вручную он занял бы несколько часов и всё равно оставался бы уязвимым для частотного анализа.

Совершенный шифр: одноразовый блокнот

По существу, классический шифр не может быть безопасным без огромного ключа, однако шифрование с огромным ключом непрактично. Тем не менее одноразовый блокнот как раз является таким шифром, причём самым безопасным из всех. В действительности он гарантирует совершенную секретность: даже если злоумышленник располагает неограниченной вычислительной мощностью, он не сможет узнать об открытом тексте ничего, кроме его длины.

В следующих разделах я покажу, как работает одноразовый блокнот, а затем в общих чертах изложу доказательство его безопасности.

Шифрование и расшифрование

Одноразовый блокнот принимает открытый текст P и случайный ключ K той же длины, что и P, и создаёт шифротекст C, определяемый как

```
C = P xor K,
```

где C, P и K - битовые строки одинаковой длины, а xor - побитовая операция исключающего ИЛИ (XOR), определяемая так: 0 xor 0 = 0, 0 xor 1 = 1, 1 xor 0 = 1, 1 xor 1 = 0.

> **Примечание.** Я представляю одноразовый блокнот в его обычной форме, работающей с битами, однако его можно приспособить и к другим символам. Например, при работе с буквами получится вариант шифра Цезаря, в котором индекс сдвига выбирается случайно для каждой буквы.

Расшифрование в одноразовом блокноте полностью совпадает с шифрованием: это всего лишь операция XOR, P = C xor K. Действительно, можно проверить, что C xor K = P xor K xor K = P, поскольку операция XOR ключа K с самим собой даёт строку из одних нулей 000...000. Вот и всё - даже проще шифра Цезаря.

Например, если P = 01101101, а K = 10110100, можно вычислить следующее:

```
C = P xor K = 01101101 xor 10110100 = 11011001.
```

При расшифровании P восстанавливается следующим вычислением:

```
P = C xor K = 11011001 xor 10110100 = 01101101.
```

Важно, что одноразовый блокнот можно использовать один раз: каждый ключ K следует применять лишь однократно. Если один и тот же K используется для шифрования P_1 и P_2 в C_1 и C_2, то перехватчик сможет вычислить:

```
C_1 xor C_2 = (P_1 xor K) xor (P_2 xor K) = P_1 xor P_2 xor K xor K = P_1 xor P_2.
```

Таким образом, перехватчик узнает XOR-разность P_1 и P_2 - сведения, которые должны оставаться секретными. Более того, если известно хотя бы одно сообщение с открытым текстом, можно восстановить второе.

Одноразовый блокнот крайне неудобен в использовании, поскольку требует ключа той же длины, что и открытый текст, и нового случайного ключа для каждого нового сообщения или набора данных. Чтобы зашифровать жёсткий диск объёмом 1 Тбайт, для хранения ключа понадобится ещё один диск объёмом 1 Тбайт! Тем не менее одноразовый блокнот применялся на протяжении всей истории: британским Управлением специальных операций во время Второй мировой войны, советскими разведчиками, Агентством национальной безопасности США (NSA); в отдельных ситуациях его используют и сегодня. (Я слышал о швейцарских банкирах, которые не смогли договориться о шифре, которому доверяли бы обе стороны, и в итоге воспользовались одноразовыми блокнотами, но не советую так поступать.)

Почему одноразовый блокнот безопасен?

Хотя одноразовый блокнот непрактичен, важно понимать, благодаря чему он безопасен. В 1940-х годах американский математик Клод Шеннон доказал: чтобы добиться совершенной секретности, ключ одноразового блокнота должен быть не короче сообщения. Идея доказательства довольно проста. Мы предполагаем, что злоумышленник обладает неограниченными возможностями и поэтому способен перебрать все ключи. Цель шифрования состоит в том, чтобы при наличии некоторого шифротекста злоумышленник не мог исключить ни один возможный открытый текст.

Интуитивное объяснение совершенной секретности одноразового блокнота таково: если K случаен, то получившийся C выглядит для злоумышленника столь же случайным, как и K, поскольку XOR случайной строки с любой фиксированной строкой даёт случайную строку. Чтобы убедиться в этом, рассмотрим вероятность получить 0 в качестве первого бита случайной строки (она равна 1/2). Какова вероятность, что результат операции XOR случайного бита со вторым битом будет равен 0? Верно, снова 1/2. Тот же аргумент можно последовательно применить к битовым строкам любой длины. Таким образом, шифротекст C выглядит случайным для злоумышленника, не знающего K, поэтому получить из C какие-либо сведения о P буквально невозможно даже злоумышленнику, располагающему неограниченным временем и вычислительной мощностью. Иными словами, знание шифротекста не сообщает об открытом тексте совершенно ничего, кроме его длины, - почти точное определение безопасного шифра.

Например, если длина шифротекста равна 128 битам (а значит, длина открытого текста также равна 128 битам), существует 2^128 возможных шифротекстов; следовательно, с точки зрения злоумышленника должно существовать 2^128 возможных открытых текстов. Но если возможных ключей меньше 2^128, злоумышленник сможет исключить некоторые открытые тексты. Например, если длина ключа составляет лишь 64 бита, злоумышленник сможет определить 2^64 возможных открытых текстов и исключить подавляющее большинство 128-битных строк. Он не узнает, чем является открытый текст, но узнает, чем он не является, а значит, секретность шифрования окажется несовершенной.

Для совершенной безопасности ключ должен быть не короче открытого текста, но при реальном применении это быстро становится непрактичным. Далее я расскажу о подходах, используемых в современном шифровании, чтобы обеспечить наилучшую одновременно возможную и практичную безопасность.

ВЕРОЯТНОСТЬ В КРИПТОГРАФИИ

Вероятность - это число, выражающее правдоподобие, или шанс, наступления некоторого события. Она задаётся числом от 0 до 1, где 0 означает «никогда», а 1 - «всегда». Чем выше вероятность, тем больше шанс. Во многих объяснениях вероятности приводится пример с белыми и красными шарами в мешке и вероятностью вытянуть шар того или иного цвета.

В криптографии вероятности часто используются для измерения шансов на успех атаки: сначала подсчитывается число успешных событий (например, событие «найти правильный секретный ключ»), а затем общее число возможных событий (например, при n-битных ключах общее число ключей равно 2^n). В этом примере вероятность того, что случайно выбранный ключ окажется правильным, равна 1/2^n: это отношение числа успешных событий (один секретный ключ) к числу возможных событий (2^n возможных ключей). Для распространённых длин ключа, таких как 128 и 256 бит, число 1/2^n пренебрежимо мало.

Вероятность того, что событие не произойдёт, равна 1 - p, где p - вероятность события. Следовательно, вероятность получить неправильный ключ в нашем примере равна 1 - 1/2^n - числу, очень близкому к 1, что означает почти полную уверенность.

Безопасность шифрования

Классические шифры небезопасны, а совершенно безопасный шифр вроде одноразового блокнота непрактичен. Поэтому, если мы хотим получить безопасные и пригодные к использованию шифры, придётся немного поступиться безопасностью. Но что в действительности означает «безопасный», помимо очевидного и неформального утверждения «перехватчики не могут расшифровать защищённые сообщения»?

Шифр безопасен, если даже при наличии большого количества пар «открытый текст - шифротекст» невозможно ничего узнать о поведении шифра при его применении к другим открытым текстам или шифротекстам. Это порождает новые вопросы:

• Как злоумышленник получает эти пары? Насколько большим должно быть «большое количество»? Всё это определяется моделями атак - предположениями о том, что злоумышленник может и чего не может делать.

• Что именно можно «узнать» и о каком «поведении шифра» идёт речь? Это определяется целями безопасности - описаниями того, что считается успешной атакой.

Модели атак и цели безопасности должны рассматриваться вместе: нельзя утверждать, что система безопасна, не объяснив, от кого или от чего она защищена. Понятие безопасности представляет собой сочетание цели безопасности и модели атаки. Мы будем говорить, что шифр удовлетворяет определённому понятию безопасности, если ни один злоумышленник, действующий в заданной модели, не способен нарушить цель безопасности.

Модели атак

Модель атаки - это набор предположений о том, как злоумышленники могут взаимодействовать с шифром и что они могут и чего не могут делать. Модель атаки преследует следующие цели:

• Установить требования для криптографов, проектирующих шифры, чтобы они знали, от каких злоумышленников и разновидностей атак следует защищаться.

• Дать пользователям ориентиры, позволяющие понять, безопасно ли применять шифр в их среде.

• Дать подсказки криптоаналитикам, пытающимся взломать шифры, чтобы они могли определить, допустима ли конкретная атака. Атака допустима только в том случае, если она осуществима в рассматриваемой модели.

Модели атак не обязаны точно совпадать с реальностью; они являются приближениями. Как выразился статистик Джордж Э. П. Бокс: «Все модели неверны; практический вопрос состоит в том, насколько неверной должна быть модель, чтобы перестать приносить пользу». Чтобы быть полезной в криптографии, модель атаки должна как минимум охватывать то, что злоумышленники действительно могут сделать для атаки на шифр. Полезно, когда модель переоценивает возможности злоумышленников, поскольку это помогает предвидеть будущие методы атак, - выживают только параноидальные криптографы. Плохая модель недооценивает злоумышленников и создаёт ложную уверенность в шифре, выставляя его безопасным в теории, хотя в действительности он небезопасен.

Принцип Керкгоффса

Во всех моделях принимается принцип Керкгоффса, согласно которому безопасность шифра должна зависеть только от секретности ключа, а не от секретности самого шифра. Сегодня, когда шифры и протоколы имеют открытые спецификации и используются всеми, это может показаться очевидным. Но исторически нидерландский лингвист Огюст Керкгоффс говорил о военных шифровальных машинах, специально созданных для определённой армии или подразделения. В своём эссе 1883 года «La Cryptographie Militaire», где он перечислил шесть требований к военной системе шифрования, Керкгоффс писал: «Система не должна требовать секретности и может быть похищена противником, не причинив неприятностей».

Модели чёрного ящика

Рассмотрим несколько полезных моделей атак, описываемых через то, что злоумышленник может наблюдать и какие запросы он может направлять шифру. Для наших целей запрос - это операция, которая передаёт входное значение некоторой функции и получает в ответ выходное значение, не раскрывая внутреннего устройства этой функции. Например, запрос шифрования принимает открытый текст и возвращает соответствующий шифротекст, не раскрывая секретный ключ.

Эти модели называются моделями чёрного ящика, потому что злоумышленник видит лишь то, что поступает в шифр и выходит из него. Например, некоторые микросхемы смарт-карт надёжно защищают как внутреннее устройство шифра, так и его ключи, но при этом к микросхеме можно подключиться и запросить расшифрование любого шифротекста. Тогда злоумышленник получит соответствующий открытый текст, который может помочь определить ключ. Это реальный пример ситуации, в которой возможны запросы расшифрования.

Существует несколько разных моделей атак на чёрный ящик. Ниже они перечислены от самой слабой к самой сильной с описанием возможностей злоумышленников в каждой модели:

• **Атаки только по шифротексту (ciphertext-only attacks, COA).** Злоумышленники наблюдают шифротексты, но не знают соответствующих открытых текстов и способа их выбора. В модели COA злоумышленники пассивны и не могут выполнять запросы шифрования или расшифрования.

• **Атаки по известному открытому тексту (known-plaintext attacks, KPA).** Злоумышленники наблюдают шифротексты и знают соответствующие открытые тексты. Таким образом, в модели KPA злоумышленник получает список пар «открытый текст - шифротекст», причём предполагается, что открытые тексты выбраны случайно. KPA - модель пассивного злоумышленника.

• **Атаки с выбранным открытым текстом (chosen-plaintext attacks, CPA).** Злоумышленники могут выполнять запросы шифрования для выбранных ими открытых текстов и наблюдать получившиеся шифротексты. Эта модель описывает ситуации, в которых злоумышленники могут выбрать весь шифруемый открытый текст или его часть, а затем увидеть шифротексты. В отличие от COA и KPA, являющихся пассивными моделями, в CPA злоумышленники активны, поскольку они воздействуют на процесс шифрования, а не просто пассивно прослушивают обмен.

• **Атаки с выбранным шифротекстом (chosen-ciphertext attacks, CCA).** Злоумышленники могут как шифровать, так и расшифровывать, то есть выполнять запросы шифрования и запросы расшифрования (для шифротекстов, отличных от целевого). Модель CCA сначала может показаться нелепой - если вы можете расшифровывать, что ещё вам нужно? Но, как и CPA, она предназначена для описания ситуаций, в которых злоумышленники способны некоторым образом повлиять на шифротекст, а позднее получить доступ к открытому тексту. Кроме того, одного расшифрования не всегда достаточно для взлома системы. Например, некоторые устройства защиты видео позволяют злоумышленникам выполнять запросы шифрования и расшифрования с

помощью микросхемы устройства, однако в такой ситуации злоумышленникам нужен ключ, чтобы распространять его дальше; тогда возможность расшифровывать «бесплатно» недостаточна для взлома системы.

В описанных моделях наблюдаемые и запрашиваемые шифротексты не возникают бесплатно. Каждый шифротекст является результатом вычисления функции шифрования. Это означает, например, что получение 2^N пар «открытый текст - шифротекст» посредством запросов шифрования требует примерно столько же вычислений, сколько перебор 2^N ключей. При оценке стоимости атаки необходимо учитывать стоимость запросов.

Модели серого ящика

В модели серого ящика злоумышленник имеет доступ к реализации шифра. Благодаря этому модели серого ящика реалистичнее моделей чёрного ящика для таких применений, как смарт-карты, встроенные и виртуализированные системы, к которым злоумышленники часто имеют физический доступ и потому способны вмешиваться во внутреннюю работу алгоритмов. По той же причине модели серого ящика сложнее определить, чем модели чёрного: они зависят от физических аналоговых свойств, а не только от входных и выходных данных алгоритма, и криптографическая теория нередко не может абстрагироваться от сложности реального мира.

Атаки по побочным каналам - семейство атак в рамках моделей серого ящика. Побочным каналом называется источник информации, зависящий от реализации шифра, будь то программная или аппаратная реализация. Злоумышленники, атакующие по побочным каналам, наблюдают или измеряют аналоговые характеристики реализации шифра, но не нарушают её целостности; такие атаки являются неинвазивными. Для чисто программных реализаций типичными побочными каналами служат время выполнения и поведение окружающей шифр системы, например сообщения об ошибках, возвращаемые значения и ветвления. В реализациях на смарт-картах типичный злоумышленник, например, измеряет энергопотребление, электромагнитное излучение или акустический шум.

Инвазивные атаки - семейство более мощных атак на реализации шифров; они дороже атак по побочным каналам, поскольку нередко требуют сложного оборудования. Базовые атаки по побочным каналам можно выполнить на обычном ПК с серийно выпускаемым осциллографом, но для инвазивных атак могут понадобиться такие средства, как микроскоп высокого разрешения и химическая лаборатория. Инвазивные атаки включают целый набор методов и процедур: применение азотной кислоты для удаления корпуса микросхемы, получение микроскопических изображений, частичную обратную разработку и изменение поведения микросхемы с помощью таких методов, как лазерная инжекция сбоев и электромагнитная инжекция.

Цели безопасности

До сих пор я неформально определял цель безопасности словами «о поведении шифра невозможно ничего узнать». Чтобы превратить эту идею в строгое математическое определение, криптографы выделяют две цели безопасности, соответствующие разным представлениям о том, что значит узнать нечто о поведении шифра:

• **Неразличимость (indistinguishability, IND).** Шифротексты должны быть неотличимы от случайных строк. Обычно это иллюстрируют воображаемой игрой: если злоумышленник выбирает два открытых текста, а затем получает шифротекст одного из них, выбранного случайно, он не должен суметь определить, какой открытый текст был зашифрован, даже выполняя запросы шифрования для этих двух открытых текстов (и запросы расшифрования, если рассматривается

модель CCA, а не CPA).

• **Неподатливость (nonmalleability, NM).** Имея шифротекст `C_1 = E(K, P_1)`, должно быть невозможно создать другой шифротекст `C_2`, соответствующий открытый текст `P_2` которого содержательно связан с `P_1` (например, создать `P_2`, равный `P_1 xor 1` или `P_1 xor X` для некоторого известного значения `X`). Удивительно, но одноразовый блокнот податлив: имея шифротекст `C_1 = P_1 xor K`, можно определить `C_2 = C_1 xor 1`, и это будет допустимый шифротекст для `P_2 = P_1 xor 1` при том же ключе `K`. Вот тебе и совершенный шифр.

Далее я рассмотрю эти цели безопасности в контексте разных моделей атак.

Понятия безопасности

Цели безопасности полезны лишь в сочетании с моделью атаки. Согласно принятому соглашению, понятие безопасности записывается как ЦЕЛЬ-МОДЕЛЬ. Например, IND-CPA обозначает неразличимость при атаке с выбранным открытым текстом, NM-CCA - неподатливость при атаке с выбранным шифротекстом и так далее. Начнём с целей безопасности в отношении злоумышленника.

Семантическая безопасность и рандомизированное шифрование: IND-CPA

Важнейшее понятие безопасности - IND-CPA, называемое также семантической безопасностью. Оно выражает интуитивную идею: пока ключ остаётся секретным, шифротексты не должны раскрывать никакой информации об открытых текстах. Чтобы обеспечить безопасность IND-CPA, шифрование должно возвращать разные шифротексты при двух вызовах для одного и того же открытого текста; в противном случае злоумышленник сможет выявить одинаковые открытые тексты по их шифротекстам, что противоречит определению, согласно которому шифротексты не должны раскрывать никакой информации. Но обратите внимание: даже схема, безопасная в смысле IND-CPA, неизбежно раскрывает один фрагмент информации об открытом тексте - его длину или хотя бы приблизительную длину. Поэтому шифровать сжатые данные, как правило, не стоит: размер сжатых данных способен раскрыть сведения об исходных данных.

Один из способов обеспечить безопасность IND-CPA - использовать рандомизированное шифрование. Как следует из названия, оно вносит случайность в процесс шифрования и при повторном шифровании одного открытого текста возвращает разные шифротексты. Тогда шифрование можно выразить как `C = E(K, R, P)`, где `R` - свежие случайные биты. Однако расшифрование остаётся детерминированным: вычислив `D(K, R, C)`, вы всегда должны получить `P` независимо от значения `R`.

Что произойдёт, если шифрование не рандомизировано? В игре IND, описанной в разделе «Цели безопасности», злоумышленник выбирает два открытых текста `P_1` и `P_2` и получает шифротекст одного из них, но не знает, какому открытому тексту он соответствует. Иными словами, злоумышленник получает `C_i = E(K, P_i)` и должен угадать, равно ли `i` единице или двум. В модели CPA он может выполнить запросы шифрования, чтобы определить и `C_1 = E(K, P_1)`, и `C_2 = E(K, P_2)`. Если шифрование не рандомизировано, достаточно проверить, равен ли `C_i` значению `C_1` или `C_2`, чтобы определить зашифрованный открытый текст и тем самым выиграть игру IND. Следовательно, рандомизация играет ключевую роль в понятии IND-CPA.

Если у вас нет генератора псевдослучайных чисел, безопасность IND-CPA всё же можно получить с помощью схемы шифрования, которой вместо случайного непредсказуемого значения требуется одноразовое число nonce (number used only once). Для каждого нового вызова шифрования nonce должно быть уникальным. Подойдёт даже простой счётчик (1, 2, 3, ...). Например, AES-CTR

(блочный шифр AES, используемый в режиме CTR) безопасен в смысле IND-CPA, если его дополнительный вход - nonce - уникален. В отличие от некоторых схем рандомизированного шифрования, где случайность лишь является частью процесса шифрования, алгоритмам с nonce это значение необходимо для расшифрования.

> **Примечание.** При рандомизированном шифровании шифротекст должен быть немного длиннее открытого текста, чтобы одному открытому тексту мог соответствовать не один шифротекст. Например, если каждому открытому тексту соответствуют 2^64 возможных шифротекстов, шифротекст должен быть как минимум на 64 бита длиннее открытого текста.

Семантически безопасное шифрование

В одной из простейших конструкций семантически безопасного шифра используется детерминированный генератор случайных битов (deterministic random bit generator, DRBG) - алгоритм, который по некоторому секретному значению возвращает биты, выглядящие случайными:

```
E(K, R, P) = (DRBG(K || R) xor P, R).
```

Здесь `R` - строка, случайно выбираемая при каждом новом шифровании и передаваемая DRBG вместе с ключом (`K || R` обозначает строку, в которой за `K` следует `R`). Этот подход напоминает одноразовый блокнот: вместо выбора случайного ключа той же длины, что и сообщение, мы используем генератор случайных битов, чтобы получить строку, выглядящую случайной.

Доказательство безопасности этого шифра в смысле IND-CPA просто, если предположить, что DRBG выдаёт случайные биты. Оно проводится от противного: если вы можете отличить шифротекст от случайной строки, то есть отличить `DRBG(K || R) xor P` от случайной строки, значит, вы можете отличить от случайной строки и `DRBG(K || R)`. Вспомним, что модель CPA позволяет получать шифротексты для выбранных значений `P`, поэтому можно выполнить XOR значения `P` со значением `DRBG(K || R) xor P` и получить `DRBG(K || R)`. Но здесь возникает противоречие, поскольку изначально мы предположили, что `DRBG(K || R)` нельзя отличить от случайной строки и что DRBG выдаёт случайные строки. Поэтому мы заключаем, что шифротексты нельзя отличить от случайных строк, а следовательно, шифр безопасен.

> **Примечание.** В качестве упражнения попробуйте определить, каким ещё понятиям безопасности удовлетворяет шифр `E(K, R, P) = (DRBG(K || R) xor P, R)`. Является ли он NM-CPA? IND-CCA? Ответы вы найдёте в следующем разделе.

Сравнение понятий безопасности

Вы узнали, что модели атак, такие как CPA и CCA, объединяются с целями безопасности, такими как NM и IND, образуя понятия безопасности NM-CPA, NM-CCA, IND-CPA и IND-CCA. Как эти понятия связаны между собой? Можно ли доказать, что удовлетворение понятию X влечёт удовлетворение понятию Y?

Некоторые связи очевидны: IND-CCA влечёт IND-CPA, а NM-CCA влечёт NM-CPA, поскольку всё, что может сделать злоумышленник CPA, способен сделать и злоумышленник CCA. Иными словами, если шифр невозможно взломать с помощью запросов с выбранным шифротекстом и выбранным открытым текстом, его невозможно взломать, выполняя лишь запросы с выбранным открытым текстом.

Менее очевидно, что IND-CPA не влечёт NM-CPA. Чтобы понять это, обратите внимание: приведённая выше конструкция IND-CPA (`DRBG(K, R) xor P, R`) не является NM-CPA. Имея шифротекст (`X, R`), можно создать шифротекст (`X xor 1, R`), являющийся допустимым шифротекстом для `P xor 1`, что противоречит понятию неподатливости.

Но обратная связь существует: NM-CPA влечёт IND-CPA. Интуитивно шифрование IND-CPA похоже на помещение предметов в мешок: вы их не видите, но можете менять их положение внутри мешка, встряхивая его вверх и вниз. NM-CPA больше похожа на сейф: поместив что-либо внутрь, вы уже не можете с ним взаимодействовать. Для IND-CCA и NM-CCA эта аналогия не работает: данные понятия эквивалентны, и каждое из них влечёт другое. Я избавлю вас от довольно технического доказательства.

ДВА ТИПА ПРИМЕНЕНИЯ ШИФРОВАНИЯ

Существует два основных типа применения шифрования. Шифрование при передаче защищает данные, отправляемые с одной машины на другую: данные шифруются перед отправкой и расшифровываются после получения, как при защищённых соединениях с сайтами электронной коммерции. Шифрование при хранении защищает данные, хранящиеся в информационной системе. Данные шифруются перед записью в память и расшифровываются перед чтением. Примерами служат системы шифрования дисков на ноутбуках, а также шифрование виртуальных машин для облачных виртуальных экземпляров. Рассмотренные нами понятия безопасности применимы к обоим типам, но выбор подходящего понятия может зависеть от конкретного применения.

Асимметричное шифрование

До сих пор мы рассматривали только симметричное шифрование, при котором две стороны совместно используют один ключ. В асимметричном шифровании ключей два: один шифрует, другой расшифровывает. Ключ шифрования называется открытым ключом и обычно считается общедоступным для всех, кто хочет отправить вам зашифрованные сообщения. Ключ расшифрования, напротив, должен оставаться секретным и называется закрытым ключом.

Открытый ключ можно вычислить по закрытому, но закрытый ключ нельзя вычислить по открытому. Иными словами, вычисление легко выполнить в одном направлении, но не в обратном - именно на этом строится криптография с открытым ключом, функции которой легко вычисляются в одном направлении, но практически необратимы.

Модели атак и цели безопасности для асимметричного шифрования примерно те же, что и для симметричного, за одним исключением: поскольку ключ шифрования открыт, любой злоумышленник может выполнять запросы шифрования, используя открытый ключ. Поэтому моделью по умолчанию для асимметричного шифрования является атака с выбранным открытым текстом.

Симметричное и асимметричное шифрование - два основных типа шифрования, которые обычно сочетают при построении защищённых систем связи. Кроме того, они лежат в основе более сложных схем, как вы увидите далее.

Когда шифры способны на большее, чем шифрование

Обычное шифрование преобразует открытые тексты в шифротексты, а шифротексты - в открытые тексты, не предъявляя никаких требований, кроме безопасности. Однако приложениям нередко нужно нечто большее: дополнительные функции безопасности или иные возможности. Поэтому криптографы создали варианты симметричного и асимметричного шифрования. Некоторые из них хорошо изучены, эффективны и широко применяются, другие остаются экспериментальными, почти не используются и отличаются низкой производительностью.

Аутентифицированное шифрование

Аутентифицированное шифрование (authenticated encryption, AE) - тип симметричного шифрования, который помимо шифротекста возвращает тег аутентификации. На рис. 1-4 показано, как аутентифицированное шифрование задаёт `AE(K, P) = (C, T)`, где тег аутентификации `T` - короткая строка, которую невозможно угадать без ключа. Расшифрование принимает `K`, `C` и `T` и возвращает открытый текст `P` только в том случае, если проверка подтверждает, что `T` является допустимым тегом для этой пары открытого текста и шифротекста; в противном случае операция прерывается и возвращает ошибку.

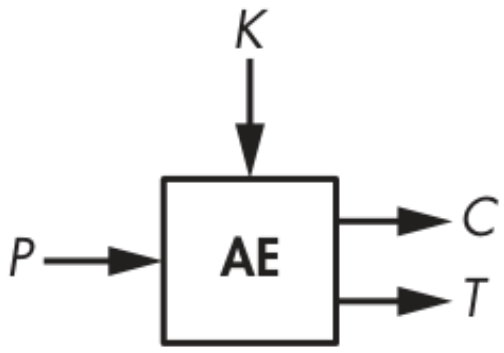

Рисунок 1-4. Аутентифицированное шифрование

Тег обеспечивает целостность сообщения и служит доказательством того, что полученный шифротекст совпадает с первоначально отправленным легитимной стороной, знающей ключ `K`. Если ключ `K` совместно используется только с одной другой стороной, тег также гарантирует, что сообщение отправлено именно этой стороной, то есть неявно подтверждает, что фактическим создателем сообщения является ожидаемый отправитель.

> **Примечание.** Здесь я говорю «создатель», а не «отправитель», потому что перехватчик может записать несколько пар `(C, T)`, отправленных стороной A стороне B, а затем вновь отправить их стороне B, выдавая себя за A. Это называется атакой повторного воспроизведения; предотвратить её можно, например, включив в сообщение значение счётчика. При расшифровании сообщения его счётчик `i` увеличивается на единицу: `i + 1`. Тогда по счётчику можно проверить, не было ли сообщение отправлено дважды, что указывало бы на попытку злоумышленника повторно воспроизвести сообщение. Это также позволяет обнаруживать потерянные сообщения.

Аутентифицированное шифрование с ассоциированными данными (authenticated encryption with associated data, AEAD) - расширение аутентифицированного шифрования, которое принимает открытые незашифрованные данные и использует их для создания тега аутентификации: `AEAD(K, P, A) = (C, A, T)`. Типичное применение AEAD - защита дейтаграмм протоколов с открытым заголовком и зашифрованной полезной нагрузкой. В таких случаях по меньшей мере часть данных заголовка должна оставаться открытой; например, адрес назначения должен быть виден, чтобы сетевые пакеты можно было маршрутизировать.

Подробнее об аутентифицированном шифровании рассказано в главе 8.

Шифрование с сохранением формата

Обычный шифр принимает биты и возвращает биты; ему безразлично, представляют ли они текст, изображение или PDF-документ. В свою очередь, шифротекст можно закодировать необработанными байтами, шестнадцатеричными символами, base64 и в других форматах. Но что делать, если шифротекст должен иметь тот же формат, что и открытый текст, как иногда требуется в системах баз данных, способных хранить данные лишь в предписанном формате?

Шифрование с сохранением формата (format-preserving encryption, FPE) решает эту проблему. Оно позволяет создавать шифротексты в том же формате, что и открытые тексты. Например, FPE может шифровать IP-адреса в IP-адреса (как показано на рис. 1-5), почтовые индексы в почтовые индексы, номера кредитных карт в номера кредитных карт с допустимой контрольной суммой и так далее.

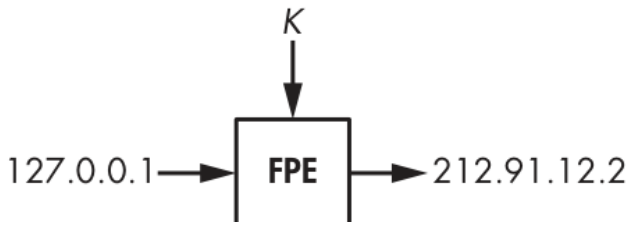

Рисунок 1-5. Шифрование IP-адресов с сохранением формата

Полностью гомоморфное шифрование

Полностью гомоморфное шифрование (fully homomorphic encryption, FHE) - святой Грааль криптографов: оно позволяет заменить шифротекст `C = E(K, P)` другим шифротекстом `C' = E(K, F(P))`, где `F(P)` может быть любой функцией от `P`, ни разу не расшифровывая исходный шифротекст `C`. Например, `P` может быть текстовым документом, а `F` - изменением части текста. Представьте облачное приложение, которое хранит ваши зашифрованные данные, но поставщик облачных услуг не знает ни содержания данных, ни характера изменений, которые вы вносите. Звучит потрясающе, не правда ли?

Но есть и обратная сторона: такое шифрование медленно - настолько медленно, что даже самая базовая операция заняла бы неприемлемо много времени. Первая схема FHE была создана в 2009 году, и с тех пор появились более эффективные варианты, однако до сих пор неясно, станет ли FHE когда-нибудь достаточно быстрым для практического применения. Вместе с тем ориентированные на конкретные приложения варианты использования (частично) гомоморфного шифрования оказались эффективнее для таких операций, как вычисление моделей машинного обучения.

Шифрование с возможностью поиска

Шифрование с возможностью поиска позволяет выполнять поиск в зашифрованной базе данных, не раскрывая искомые выражения, поскольку сам поисковый запрос также шифруется. Подобно полностью гомоморфному шифрованию, такое шифрование могло бы повысить конфиденциальность многих облачных приложений, скрывая ваши поисковые запросы от поставщика облачных услуг. Некоторые коммерческие решения заявляют о поддержке шифрования с возможностью поиска, хотя в основном они построены на стандартной криптографии с несколькими приёмами, обеспечивающими частичную возможность поиска. Однако на момент написания книги шифрование с возможностью поиска остаётся экспериментальным направлением в исследовательском сообществе.

Настраиваемое шифрование

Настраиваемое шифрование (tweakable encryption, TE) похоже на обычное, но использует дополнительный параметр, называемый настройкой (tweak), который предназначен для имитации разных версий шифра (см. рис. 1-6). Настройкой может служить уникальное значение для каждого клиента, гарантирующее, что шифр клиента нельзя клонировать другим сторонам, использующим тот же продукт, но основное применение TE - шифрование дисков. Впрочем, TE не привязано к одному применению: это низкоуровневый тип шифрования, используемый при построении других схем, например режимов аутентифицированного шифрования.

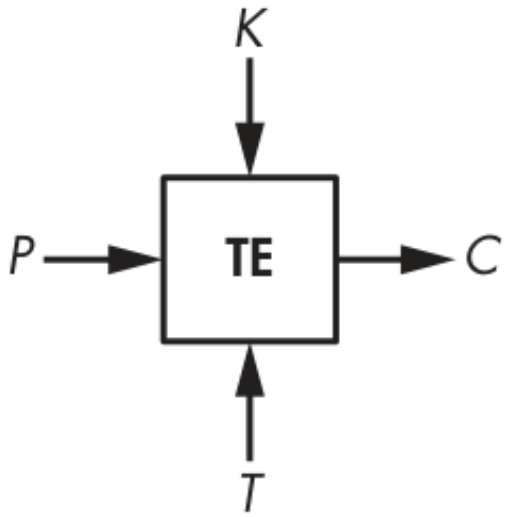

Рисунок 1-6. Настраиваемое шифрование

При шифровании дисков TE шифрует содержимое устройств хранения, таких как жёсткие или твердотельные диски. (Рандомизированное шифрование использовать нельзя, поскольку оно увеличивает объём данных, что неприемлемо для файлов на носителе.) Чтобы сделать шифрование непредсказуемым, TE использует значение настройки, зависящее от положения шифруемых данных, - обычно это номер сектора или индекс блока.

Что может пойти не так

Алгоритмы шифрования или их реализации могут не обеспечить конфиденциальность по множеству причин. Проблема может состоять в несоответствии требованиям безопасности (таким как «быть безопасным в смысле IND-CPA») либо в выборе требований, не соответствующих действительности (если вы ориентируетесь лишь на безопасность IND-CPA, тогда как злоумышленники в действительности могут выполнять запросы с выбранным шифротекстом). Увы, многие инженеры вообще не задумываются о требованиях криптографической безопасности и хотят получить «безопасность», не понимая, что именно это значит. Обычно это верный путь к катастрофе. Рассмотрим два примера.

Слабый шифр

Первый пример касается шифров, которые можно атаковать методами криптоанализа, как произошло со стандартом мобильной связи 2G. Для шифрования в мобильных телефонах 2G применялся шифр A5/1, оказавшийся слабее, чем ожидалось, вследствие чего любой человек с подходящими навыками и инструментами мог перехватывать звонки. Операторам связи пришлось искать обходные способы предотвращения атаки.

> **Примечание.** Стандарт 2G также определял A5/2 - шифр для регионов за пределами Европейского союза и США. A5/2 был намеренно ослаблен, чтобы не допустить повсеместного применения сильного шифрования.

Тем не менее атаковать A5/1 непросто, и исследователям потребовалось более десяти лет, чтобы разработать эффективный метод криптоанализа. Более того, эта атака представляет собой компромисс «время - память» (time-memory trade-off, TMTO) - метод, при котором сначала в течение нескольких дней или недель выполняются вычисления для построения больших таблиц поиска, а затем эти таблицы используются непосредственно в атаке. Для A5/1 объём предварительно вычисленных таблиц составляет около 1 Тбайт. Более поздние стандарты мобильного шифрования, такие как 3G и LTE, определяют более сильные шифры, однако это не означает, что их шифрование не будет скомпрометировано; это означает лишь, что его не скомпрометируют путём взлома симметричного шифра, входящего в систему.

Неверная модель

Следующий пример касается недействительной модели атаки, в которой не были учтены некоторые побочные каналы.

Многие протоколы связи с шифрованием следят за тем, чтобы использовать шифры, считающиеся безопасными в модели CPA или CCA. Однако для некоторых атак не требуются ни запросы шифрования, как в модели CPA, ни запросы расшифрования, как в модели CCA. Им достаточно запросов проверки допустимости, которые сообщают, является ли шифротекст допустимым; такие запросы обычно отправляются системе, отвечающей за расшифрование шифротекстов. Примером служат атаки на оракул заполнения, в ходе которых злоумышленник узнаёт, соответствует ли шифротекст требуемому формату.

Если говорить конкретно об атаках на оракул заполнения, шифротекст допустим только тогда, когда его открытый текст имеет правильное заполнение - последовательность байтов, добавленную к

открытому тексту для упрощения шифрования. При неправильном заполнении расшифрование завершается ошибкой; злоумышленники часто могут обнаруживать такие ошибки и пытаться ими воспользоваться. Например, наличие исключения Java `javax.crypto.BadPaddingException` указывает на обнаружение неправильного заполнения.

В 2010 году исследователи обнаружили атаки на оракул заполнения в нескольких серверах веб-приложений. Запросы проверки допустимости заключались в отправке шифротекста некоторой системе и наблюдении за тем, выдаст ли она ошибку. Благодаря этим запросам исследователи могли расшифровывать в остальном безопасные шифротексты, не зная ключа.

Криптографы часто упускают из виду атаки вроде атак на оракул заполнения, поскольку обычно они зависят от поведения приложения и от того, как пользователи могут с ним взаимодействовать. Но если не предусмотреть такие атаки и не включить их в модель при проектировании и развёртывании криптографии, вас могут ждать неприятные сюрпризы.

Дополнительная литература

На протяжении всей книги мы подробнее рассматриваем шифрование и его различные формы, уделяя особое внимание работе современных безопасных шифров. И всё же я не могу охватить абсолютно всё и обошёл стороной множество увлекательных тем. Например, чтобы изучить теоретические основы шифрования и глубже понять понятие неразличимости, прочитайте статью 1982 года, в которой была введена идея семантической безопасности: «Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information» Голдвассер и Микали. Если вас интересуют физические атаки и криптографическое оборудование, главным источником служат материалы конференции Cryptographic Hardware and Embedded Systems (CHES).

Существует и множество других типов шифрования, не рассмотренных в этой главе: шифрование на основе атрибутов, широковещательное шифрование, функциональное шифрование, шифрование на основе идентификационных данных, шифрование с привязкой к сообщению и прокси-перешифрование - вот лишь несколько примеров. С новейшими исследованиями по этим темам можно ознакомиться в электронном архиве криптографических научных работ https://eprint.iacr.org.

Глава 2. Случайность

Случайность встречается в криптографии повсюду: при создании секретных ключей, в схемах шифрования и даже в атаках на криптосистемы. Без случайности криптография была бы невозможна, поскольку все операции стали бы предсказуемыми, а следовательно, небезопасными.

В этой главе вводится понятие случайности в контексте криптографии и её применений. Мы обсудим генераторы псевдослучайных чисел и способы получения операционными системами надёжной случайности, а в заключение рассмотрим реальные примеры того, как дефектная случайность влияет на безопасность.

Случайно или неслучайно?

Вероятно, вы уже слышали выражение «случайные биты», однако, строго говоря, последовательности случайных битов не существует. Случайным является алгоритм, или процесс, создающий последовательность случайных битов; поэтому, говоря «случайные биты», мы в действительности имеем в виду случайно сгенерированные биты.

Как выглядят случайные биты? Например, 8-битная строка `11010110` может казаться более случайной, чем `00000000`, хотя вероятность генерации каждой из них одинакова и равна 1/256. Значение `11010110` выглядит более случайным, чем `00000000`, поскольку обладает признаками, характерными для случайно сгенерированного значения: в `11010110` нет очевидной закономерности.

Когда мы видим строку `11010110`, наш мозг отмечает, что она содержит три нуля и пять единиц, как и ещё 55 восьмибитных строк (`11111000`, `11110100`, `11110010` и так далее), тогда как строка из восьми нулей существует только одна. Поскольку сочетание «три нуля и пять единиц» встречается вероятнее, чем сочетание «восемь нулей», мы считаем `11010110` случайной, а `00000000` - неслучайной, хотя в действительности это не так.

Этот пример иллюстрирует два типа ошибок, которые люди часто совершают при распознавании случайности:

• **Принять неслучайность за случайность.** Считать, что объект создан случайно, лишь потому, что он выглядит случайным.

• **Принять случайность за неслучайность.** Считать, что случайно возникшие закономерности появились по какой-то иной причине, а не случайно.

Различие между тем, что выглядит случайным, и тем, что действительно случайно, имеет решающее значение. В криптографии неслучайность зачастую фактически означает небезопасность.

Выражение «это произошло случайно» отражает свойство, благодаря которому из сложной системы - в данном случае нашей Вселенной, подчиняющейся законам физики, детерминированной на макроскопическом уровне и истинно случайной на субатомном, квантовом уровне, - могут возникать конкретные закономерности, например строка `00000000`. Согласно закону больших чисел, если происходит много событий, некоторые из них не будут выглядеть случайными - например, последовательность идущих подряд чисел в лотерейном тираже. Многие псевдонауки и системы убеждений на самом деле представляют собой случаи, когда случайность принимают за неслучайность.

Случайность как распределение вероятностей

Любой рандомизированный процесс характеризуется распределением вероятностей, которое сообщает всё, что можно знать о случайности этого процесса. Распределение вероятностей, или просто распределение, перечисляет исходы рандомизированного процесса и ставит каждому исходу в соответствие вероятность.

Вероятность измеряет правдоподобие наступления события. Она выражается действительным числом от 0 до 1, где вероятность 0 означает невозможность, а вероятность 1 - достоверность. Например, при броске двухсторонней монеты вероятность выпадения каждой стороны вверх равна 1/2 (или 0,5), а вероятность того, что монета останется стоять на ребре, близка к 0.

Распределение вероятностей должно включать все возможные исходы, а сумма всех вероятностей должна равняться 1. В частности, если существует `N` возможных событий, им соответствуют `N` вероятностей `p_1, p_2, ..., p_N`, причём `p_1 + p_2 + ... + p_N = 1`. При броске монеты распределение имеет значение 1/2 для орла и 1/2 для решки. Сумма обеих вероятностей равна `1/2 + 1/2 = 1`, поскольку монета упадёт одной из двух сторон вверх.

Равномерным называется распределение, в котором все вероятности равны, то есть все исходы равновероятны. Если существует `N` событий, вероятность каждого из них равна `1/N`. Например, если 128-битный ключ выбирается равномерно случайным образом, то есть согласно равномерному распределению, вероятность каждого из `2^128` возможных ключей должна быть равна `1/2^128`.

Напротив, в неравномерном распределении вероятности не все равны. Бросок монеты с неравномерным распределением называется смещённым: например, орёл может выпадать с вероятностью 1/4, а решка - с вероятностью 3/4.

> **Примечание.** С помощью утяжелённой игральной кости можно жульничать, сделав вероятности выпадения каждой из шести граней отличными от 1/6; однако создать смещённую монету нельзя. Бросок монеты может быть смещён лишь в том случае, если «монете позволяют отскакивать или вращаться, а не просто подбрасывают её в воздух», как описано в статье «You Can Load a Die but You Can't Bias a Coin».

Энтропия: мера неопределённости

Энтропия - мера неопределённости, или беспорядка, в системе. Чем выше энтропия, тем меньше определённости в результате рандомизированного процесса.

Энтропию распределения вероятностей можно вычислить. Если распределение состоит из вероятностей `p_1, p_2, ..., p_N`, его энтропия равна отрицательной сумме произведений всех вероятностей на их логарифмы:

```
-p_1 * log(p_1) - p_2 * log(p_2) - ... - p_N * log(p_N).
```

Здесь функция `log` означает двоичный логарифм, то есть логарифм по основанию два. В отличие от натурального логарифма, двоичный выражает информацию в битах и даёт целые значения, когда вероятности являются степенями двойки. Например, `log(1/2) = -1`, `log(1/4) = -2`, а в общем случае `log(1/2^n) = -n`. (Отрицательная сумма берётся как раз для того, чтобы получить положительное число.) Поэтому случайные 128-битные ключи, созданные с равномерным распределением, имеют следующую энтропию:

```
2^128 * (-2^(-128) * log(2^(-128))) = -log(2^(-128)) = 128 бит.
```

Если заменить 128 любым целым числом n, энтропия равномерно распределённой n-битной строки составит n бит.

Энтропия максимальна при равномерном распределении, поскольку такое распределение обеспечивает максимальную неопределённость: ни один исход не вероятнее остальных. Следовательно, n-битные значения не могут обладать энтропией выше n бит.

Точно так же при неравномерном распределении энтропия ниже. Рассмотрим пример с броском монеты. Энтропия честного броска равна:

```
-(1/2) * log(1/2) - (1/2) * log(1/2) = 1/2 + 1/2 = 1 бит.
```

Что произойдёт, если вероятность выпадения одной стороны монеты выше, чем другой? Предположим, вероятность орла равна 1/4, а решки - 3/4. (Помните, что сумма всех вероятностей должна равняться 1.)

Энтропия такого смещённого броска равна:

```
-(3/4) * log(3/4) - (1/4) * log(1/4)
~= -(3/4) * (-0,415) - (1/4) * (-2)
~= 0,81 бита.
```

Тот факт, что 0,81 меньше энтропии честного броска, равной 1 биту, показывает: чем сильнее смещена монета, тем менее равномерно распределение и тем ниже энтропия. Если продолжить этот пример, при вероятности орла 1/10 энтропия составит 0,469, а при снижении вероятности до 1/100 энтропия упадёт до 0,081.

> **Примечание.** Энтропию можно также рассматривать как меру информации. Например, результат честного броска монеты даёт ровно 1 бит информации - орёл или решка, - а заранее предсказать результат вы не можете. В случае нечестной монеты вы заранее знаете, что решка вероятнее, поэтому способны предсказать исход. Результат броска нечестной монеты даёт вам информацию, необходимую, чтобы предсказать результат с уверенностью.

Генераторы случайных и псевдослучайных чисел

Для безопасности криптосистемам нужна случайность, а значит, им необходим компонент, из которого эту случайность можно получать. Задача такого компонента - по запросу возвращать случайные биты. Для их генерации нужны две вещи:

• Источник энтропии, предоставляемый генераторами случайных чисел.

• Криптографический алгоритм, создающий из источника энтропии высококачественные случайные биты. Такой алгоритм реализуют генераторы псевдослучайных чисел.

Совместное использование генераторов случайных и псевдослучайных чисел - ключ к практичной и безопасной криптографии. Сначала кратко рассмотрим работу генераторов случайных чисел, а затем подробно разберём генераторы псевдослучайных чисел.

Случайность поступает из окружающей среды, которая аналогова, хаотична, неопределённа, а потому непредсказуема. Одни лишь компьютерные алгоритмы не могут создавать случайность. В криптографии она обычно поступает от генераторов случайных чисел (random number generators, RNG) - программных или аппаратных компонентов, которые используют энтропию аналогового мира для получения непредсказуемых битов в цифровой системе. Например, RNG может непосредственно выбирать биты из измерений температуры, акустического шума, турбулентности воздуха или статического электричества. К сожалению, такие аналоговые источники энтропии

доступны не всегда, а их энтропию зачастую трудно оценить.

RNG может также собирать энтропию в работающей операционной системе, получая её от подключённых датчиков, устройств ввода-вывода, сетевой или дисковой активности, системных журналов, выполняющихся процессов и действий пользователя, например нажатий клавиш и перемещений мыши. Такая активность, порождённая системой и человеком, может быть хорошим источником энтропии, но она нестабильна и поддаётся манипуляциям злоумышленника. Кроме того, случайные биты из неё поступают медленно.

> **Примечание.** Квантовые генераторы случайных чисел (quantum random number generators, QRNG) - разновидность RNG, использующая случайность, возникающую в квантово-механических явлениях, например радиоактивном распаде, поляризации фотонов или тепловом шуме. Эти явления, не описываемые уравнениями, которые определяют будущее состояние по текущему, случайны в абсолютном смысле. Однако на практике необработанные биты, извлечённые из QRNG, могут быть смещёнными и обычно создаются медленно. Как и упомянутые выше источники энтропии, они требуют постобработки, чтобы с высокой скоростью получать надёжные биты.

Генераторы псевдослучайных чисел (pseudorandom number generators, PRNG) решают проблему получения случайности, надёжно создавая множество искусственных случайных битов из небольшого количества истинно случайных. Например, RNG, преобразующий движения мыши в случайные биты, прекратит работу, если вы перестанете двигать мышь, тогда как PRNG по запросу всегда возвращает псевдослучайные биты.

PRNG опираются на RNG, но ведут себя иначе: RNG сравнительно медленно получают истинно случайные биты из аналоговых источников, недетерминированным образом, без гарантии равномерного распределения или высокой энтропии каждого бита. В противоположность им PRNG быстро создают из цифровых источников биты, которые выглядят случайными, причём делают это детерминированно, с равномерным распределением и гарантированно достаточно высокой для криптографических применений энтропией. По существу, PRNG преобразуют небольшое количество ненадёжных случайных битов в длинный поток надёжных псевдослучайных битов, пригодных для криптографии, как показано на рис. 2-1.

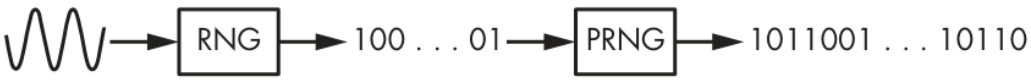

Рисунок 2-1. RNG создают небольшое количество ненадёжных битов из аналоговых источников, а PRNG разворачивают эти биты в длинный поток надёжных битов

Как работают PRNG

PRNG через регулярные промежутки времени получает случайные биты от RNG и использует их для обновления содержимого большого буфера памяти, называемого пулом энтропии. Пул энтропии служит для PRNG источником энтропии подобно тому, как физическая среда служит источником для RNG. Обновляя пул энтропии, PRNG перемешивает его биты, помогая устранить любое статистическое смещение.

Чтобы создать псевдослучайные биты, PRNG выполняет алгоритм детерминированного генератора случайных битов (deterministic random bit generator, DRBG), который разворачивает несколько битов из пула энтропии в гораздо более длинную последовательность. Как следует из названия, DRBG детерминирован, а не случаен: при одинаковых входных данных он всегда выдаёт одинаковый

результат. PRNG следит за тем, чтобы его DRBG никогда не получал одни и те же входные данные дважды и мог создавать уникальные псевдослучайные последовательности.

В ходе работы PRNG выполняет три операции:

• `init()`. Инициализирует пул энтропии и внутреннее состояние PRNG.

• `refresh(R)`. Обновляет пул энтропии некоторыми данными `R`, обычно полученными от RNG.

• `next(N)`. Возвращает `N` псевдослучайных битов и обновляет пул энтропии.

Операция `init` сбрасывает PRNG в новое состояние, повторно инициализирует пул энтропии некоторым значением по умолчанию, а также инициализирует все переменные и буферы памяти, которые PRNG использует для выполнения операций `refresh` и `next`.

Операцию `refresh` часто называют повторным засевом, а её аргумент `R` - начальным значением, или зерном (seed). Когда RNG недоступен, зерном могут служить уникальные значения, жёстко заданные в системе. Операцию `refresh` обычно вызывает операционная система, а `next` обычно вызывают или запрашивают приложения. Операция `next` запускает DRBG и изменяет пул энтропии, чтобы следующий вызов выдал другие псевдослучайные биты.

Вопросы безопасности

Кратко рассмотрим, как PRNG решают вопросы безопасности высокого уровня. В частности, PRNG должны гарантировать устойчивость к восстановлению прошлого и устойчивость к предсказанию. Устойчивость к восстановлению прошлого (называемая также прямой секретностью) означает невозможность восстановить ранее созданные биты, а устойчивость к предсказанию (обратная секретность) - невозможность предсказать будущие биты.

Чтобы обеспечить устойчивость к восстановлению прошлого, PRNG должен гарантировать необратимость преобразований, выполняемых при обновлении состояния операциями `refresh` и `next`. Тогда, если злоумышленник скомпрометирует систему и получит значение пула энтропии, он не сможет определить предыдущие значения пула или ранее созданные биты. Для устойчивости к предсказанию PRNG должен регулярно вызывать `refresh` со значениями `R`, неизвестными злоумышленнику и трудно угадываемыми; это не позволит злоумышленнику определять будущие значения пула энтропии, даже если весь пул скомпрометирован. (Если бы список значений `R` был известен, для воссоздания пула понадобилось бы знать порядок вызовов `refresh` и `next`.)

PRNG Fortuna

Fortuna - конструкция PRNG, применяемая в Windows; первоначально её разработали в 2003 году Нильс Фергюсон и Брюс Шнайер. Fortuna пришла на смену Yarrow - разработке Джона Келси и Брюса Шнайера 1998 года, которая долго использовалась в операционных системах macOS и iOS, но затем также была заменена на Fortuna. Я не стану приводить здесь спецификацию Fortuna или показывать её реализацию, но постараюсь объяснить принцип работы. Полное описание Fortuna вы найдёте в главе 9 книги Фергюсона, Шнайера и Коно «Cryptography Engineering» (Wiley, 2010).

Внутренняя память Fortuna содержит следующее:

• 32 пула энтропии `P_1, P_2, ..., P_32`, причём `P_i` используется при каждом `2^i`-м повторном засеве.

• Ключ `K` и счётчик `C` (оба по 16 байт). Вместе они образуют внутреннее состояние DRBG в Fortuna.

В самом упрощённом виде Fortuna работает так:

• `init()` присваивает `K` и `C` нулевые значения и очищает 32 пула энтропии `P_i`, где `i = 1 ... 32`.

• `refresh(R)` добавляет данные `R` в один из пулов энтропии. Система выбирает RNG, используемые для создания значений `R`, и должна регулярно вызывать `refresh`.

• `next(N)` обновляет `K` с помощью данных из одного или нескольких пулов энтропии; выбор пулов главным образом зависит от числа уже выполненных обновлений `K`. Затем запрошенные `N` битов создаются шифрованием `C` с ключом `K`. Если одного шифрования `C` недостаточно, Fortuna шифрует `C + 1`, затем `C + 2` и так далее, пока не получит достаточно битов.

Хотя операции Fortuna выглядят довольно просто, правильно реализовать их трудно. Прежде всего необходимо безошибочно воспроизвести все детали алгоритма: способ выбора пулов энтропии, тип шифра для `next`, поведение при отсутствии поступающей энтропии и так далее. Спецификация определяет большинство подробностей, однако не содержит полного набора тестов для проверки правильности реализации, поэтому трудно убедиться, что ваша реализация Fortuna будет вести себя ожидаемым образом.

Даже при правильной реализации Fortuna нарушения безопасности возможны по причинам, не связанным с неверным алгоритмом. Например, Fortuna может не заметить, что RNG создаёт недостаточно случайных битов, и вследствие этого начнёт выдавать псевдослучайные биты более низкого качества или вовсе перестанет их выдавать.

Ещё один риск, присущий реализациям Fortuna, связан с возможностью раскрытия злоумышленникам файлов начальных значений. Данные в таких файлах используются для передачи энтропии в Fortuna через вызовы `refresh`, когда RNG недоступен немедленно, например сразу после перезагрузки системы, прежде чем системные RNG зарегистрировали какие-либо непредсказуемые события. Однако если один и тот же файл начальных значений использовать дважды, Fortuna дважды создаст одну и ту же последовательность битов. Поэтому после использования файлы начальных значений следует удалять, чтобы исключить их повторное применение.

Наконец, если два экземпляра Fortuna находятся в одинаковом состоянии, поскольку совместно используют один файл начальных значений (то есть во всех пулах энтропии находятся одинаковые данные, включая `C` и `K`), операция `next` вернёт одинаковые биты в обоих экземплярах.

Криптографические и некриптографические PRNG

PRNG бывают криптографическими и некриптографическими. Некриптографические PRNG предназначены для создания равномерных распределений в таких приложениях, как научное моделирование или видеоигры. Однако использовать некриптографические PRNG в криптографических приложениях нельзя: они небезопасны, поскольку учитывают лишь качество распределения вероятностей битов, но не их предсказуемость. Криптографические PRNG, напротив, непредсказуемы, так как в них учитывается также стойкость базовых операций, используемых для получения хорошо распределённых битов.

К сожалению, большинство PRNG, предоставляемых языками программирования, - например, `rand` и `drand48` из libc, `rand` и `mt_rand` из PHP, модуль `random` в Python и класс `java.util.Random` в Java - являются некриптографическими. Выбор некриптографического PRNG по умолчанию - верный путь к катастрофе, потому что его нередко в итоге используют в криптографических приложениях. Поэтому для создания случайности, связанной с криптографией или безопасностью, обязательно применяйте только криптографические PRNG.

Популярный некриптографический PRNG: Mersenne Twister

Алгоритм Mersenne Twister (MT) - некриптографический PRNG, который на момент написания книги используется в PHP, Python, R, Ruby и многих других системах. Его даже применяли, к сожалению, в генераторах ключей блокчейн-кошельков. MT создаёт равномерно распределённые случайные биты без статистического смещения, но он предсказуем: имея несколько битов, созданных MT, можно угадать, какие биты последуют далее.

Заглянем внутрь и посмотрим, почему Mersenne Twister небезопасен. Алгоритм MT значительно проще криптографических PRNG: его внутреннее состояние представляет собой массив `S` из 624 32-битных слов. Изначально этот массив равен `S_1, S_2, ..., S_624`, затем изменяется на `S_2, ..., S_625`, потом на `S_3, ..., S_626` и так далее согласно уравнению:

```
S_(k+624) = S_(k+397) xor A((S_k and 0x80000000) or (S_(k+1) and 0x7fffffff)).
```

Здесь `xor` обозначает побитовую операцию XOR (`^` в языке C), `and` - побитовую операцию AND (`&` в C), `or` - побитовую операцию OR (`|` в C), а `A` - функцию, преобразующую некоторое 32-битное слово `x` в `(x >> 1)`, если старший бит `x` равен 0, и в `(x >> 1) xor 0x9908b0df` в противном случае.

В этом уравнении биты `S` взаимодействуют друг с другом только посредством XOR. Операторы `and` и `or` никогда не объединяют два бита `S` между собой, а лишь объединяют биты `S` с битами констант `0x80000000` и `0x7fffffff`. Поэтому любой бит `S_625` можно выразить как XOR битов `S_398`, `S_1` и `S_2`, а любой бит любого будущего состояния - как XOR-комбинацию битов из начального состояния `S_1, ..., S_624`. (Когда, например, `S_(228+624) = S_852` выражается как функция от `S_625`, `S_228` и `S_229`, значение `S_625`, в свою очередь, можно заменить его выражением через `S_398`, `S_1` и `S_2`.)

Поскольку начальное состояние содержит ровно `624 * 32 = 19 968` битов (или 624 32-битных слова), любой выходной бит можно выразить уравнением не более чем из 19 969 членов (19 968 битов и одного постоянного бита). Это около 2,5 Кбайт данных. Верно и обратное: биты начального состояния можно выразить как XOR выходных битов.

Небезопасность линейности

XOR-комбинацию битов называют линейной комбинацией. Например, если `X`, `Y` и `Z` - биты, выражение `X xor Y xor Z` является линейной комбинацией, а `(X and Y) xor Z` - нет, поскольку содержит AND (`and`). Если инвертировать бит `X` в `X xor Y xor Z`, результат также изменится независимо от значений `Y` и `Z`. Напротив, если инвертировать бит `X` в `(X and Y) xor Z`, результат изменится только тогда, когда бит `Y` в той же позиции равен 1. Суть в том, что линейные комбинации предсказуемы: не нужно знать значения битов, чтобы предсказать, как изменение их значений повлияет на результат.

Для сравнения: если бы алгоритм MT был криптографически стойким, его уравнения были бы нелинейными и включали бы не только отдельные биты, но и их AND-комбинации (произведения), например `S_1S_15S_182` или `S_17S_256S_257S_354S_498S_601`. Хотя линейные комбинации этих битов содержат не более 624 переменных, нелинейные комбинации допускают до `2^624` переменных. Полную систему таких уравнений было бы невозможно не только решить, но даже записать. (Заметьте, что `2^305` - значительно меньшее число - считается оценкой информационной ёмкости наблюдаемой Вселенной.)

Главное здесь то, что линейные преобразования приводят к коротким уравнениям, размер которых сопоставим с числом переменных и которые легко решаются, тогда как нелинейные преобразования порождают уравнения экспоненциального размера, практически неразрешимые. Поэтому задача криптографов - создавать алгоритмы PRNG, которые имитируют столь сложные

нелинейные преобразования с помощью небольшого числа простых операций.

> **Примечание.** Линейность - лишь один из множества критериев безопасности. Нелинейность необходима, но сама по себе не делает PRNG криптографически безопасным.

Бесполезность статистических тестов

Наборы статистических тестов, такие как TestU01, Diehard или набор Национального института стандартов и технологий США (NIST), являются одним из способов проверки качества псевдослучайных битов. Эти тесты принимают образец созданных PRNG псевдослучайных битов (например, объёмом 1 Мбайт), вычисляют статистические показатели распределения определённых битовых шаблонов и сравнивают результаты с типичными результатами для равномерного распределения. Например, некоторые тесты подсчитывают количество единичных и нулевых битов или распределение 8-битных шаблонов. Но статистические тесты по большей части не имеют отношения к криптографической безопасности: можно создать криптографически слабый PRNG, который обманет любой статистический тест.

Запуская статистические тесты на случайно сгенерированных данных, вы обычно получите множество статистических показателей. Как правило, это p-значения - распространённый статистический показатель. Результаты не всегда легко истолковать, поскольку они редко сводятся к простому «пройдено» или «не пройдено». Если первые результаты кажутся аномальными, не беспокойтесь: причиной может быть случайное отклонение либо слишком малое число проверяемых образцов. Чтобы убедиться, что наблюдаемые результаты нормальны, сравните их с результатами для некоторого надёжного образца того же размера, например созданного набором OpenSSL следующей командой:

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

PRNG в реальном мире

Теперь обратимся к реализации PRNG в реальном мире. Криптографические PRNG присутствуют в операционных системах большинства платформ: от настольных компьютеров и ноутбуков до встроенных систем, таких как маршрутизаторы и телевизионные приставки, а также в виртуальных машинах, мобильных телефонах и так далее. Большинство этих PRNG программные, однако чисто аппаратные PRNG используются приложениями, работающими в ОС, а иногда и другими PRNG, выполняющимися поверх криптографических библиотек или приложений.

Далее мы рассмотрим наиболее широко распространённые PRNG: применяемые в Linux, Android и многих других системах на основе Unix; используемые в Windows; а также аппаратный PRNG в современных микропроцессорах Intel.

Случайные биты в Linux

Файл устройства `/dev/urandom` - интерфейс пользовательского пространства к криптографическому PRNG в операционных системах на основе ядра Linux. Обычно именно его используют для создания надёжных случайных битов. Поскольку это файл устройства, случайные биты запрашиваются у `/dev/urandom` путём чтения файла. Например, следующая команда использует `/dev/urandom`, чтобы записать 10 Мбайт случайных битов в файл:

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

Неправильный способ использования /dev/urandom

Можно написать наивную и небезопасную программу на C, подобную показанной в листинге 2-1, прочитать случайные биты и надеяться на лучшее, но это плохая идея.

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

Листинг 2-1. Небезопасное использование `/dev/urandom`

Этот код небезопасен; он даже не проверяет возвращаемые значения `open()` и `read()`, поэтому буфер, в котором ожидаются случайные данные, может оказаться заполненным нулями или остаться неизменённым.

Более безопасный способ использования /dev/urandom

В листинге 2-2, скопированном из библиотеки LibreSSL, показан более безопасный способ использования `/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 = open("/dev/urandom", flags, 0);                    /* 1 */
    if (fd == -1) {
        if (errno == EINTR)
            goto start;
        goto nodevrandom;
    }
#ifndef O_CLOEXEC
    fcntl(fd, F_SETFD, fcntl(fd, F_GETFD) | FD_CLOEXEC);
#endif

    /* Слегка проверяем, что узел устройства выглядит разумно. */
    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 = read(fd, (char *)buf + i, wanted);     /* 2 */

        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; /* Требование выполнено */
    }

nodevrandom:
    errno = EIO;
    return -1;
}
```

Листинг 2-2. Безопасное использование `/dev/urandom`

В отличие от листинга 2-1, в листинге 2-2 выполняется несколько проверок разумности. Сравните, например, вызовы `open()` **1** и `read()` **2** с их аналогами в листинге 2-1: более безопасный код проверяет возвращаемые значения этих функций, а при сбое закрывает файловый дескриптор и возвращает -1.

Различия между /dev/urandom и /dev/random до 2022 года

PRNG Linux, определённый в файле `drivers/char/random.c` ядра Linux, в 2022 году претерпел значительные изменения, начиная с версии ядра 5.17.

Прежде всего общая структура PRNG, сходная в старых и новых версиях, основана на сборе энтропии из различных источников (включая системную активность, например работу с клавиатурой, мышью и дисками), а также на пуле энтропии, который можно представить как большой массив, заполняемый хешированными данными, собранными из источников энтропии. Затем DRBG создаёт потоки псевдослучайных данных, возвращаемые при чтении `/dev/random` или `/dev/urandom` либо при системном вызове `getrandom()`.

Исторически, до версии ядра 5.17, PRNG Linux вёл себя следующим образом: в отличие от `/dev/urandom`, интерфейс `/dev/random` был блокирующим. Если, по оценке ядра, уровень энтропии PRNG был недостаточен, `/dev/random` при чтении переставал возвращать байты, то есть «блокировался», пока ядро не оценивало уровень энтропии как достаточный. Это была неудачная идея. Во-первых, оценки энтропии печально известны своей ненадёжностью и могут быть обмануты злоумышленниками (это одна из причин, по которым Fortuna отказалась от оценки энтропии, применявшейся в Yarrow). Кроме того, оценённая энтропия `/dev/random` довольно быстро исчерпывалась, что могло приводить к отказу в обслуживании, замедляя приложения, вынужденные ожидать дополнительной энтропии. В итоге на практике `/dev/random` не превосходил `/dev/urandom` и создавал больше проблем, чем решал.

Различия между /dev/urandom и /dev/random с 2022 года

В версии ядра Linux 2022 года (5.17 и новее) было внесено несколько улучшений. Во-первых, при формировании содержимого пула хеш-функцию SHA-1 заменили на BLAKE2. Самое значительное изменение коснулось относительного поведения `/dev/random` и `/dev/urandom`; предлагалось даже полностью устранить различия между ними. На момент написания книги на большинстве платформ оба интерфейса определяют, достаточно ли энтропии, но `/dev/urandom` возобновляет создание псевдослучайных битов, если ядру не удаётся собрать достаточно энтропии, тогда как `/dev/random` блокируется.

Кроме того, логика оценки энтропии в ядре была существенно улучшена: вместо предположения, что при чтении битов PRNG энтропия уменьшается (криптографическая бессмыслица), ядро просто отслеживает момент накопления достаточной неопределённости, то есть энтропии, например при запуске системы.

Значение энтропии системы Linux можно прочитать в файле `/proc/sys/kernel/random/entropy_avail`. В старых версиях ядра это значение имело максимум 4096 бит и уменьшалось при создании битов PRNG. В новых ядрах оно ограничено 256 битами и потому больше не уменьшается.

Функция CryptGenRandom() в Windows

В Windows устаревшим интерфейсом пользовательского пространства к системному PRNG является функция `CryptGenRandom()` из интерфейса прикладного программирования Cryptography API. В современных версиях Windows функция `CryptGenRandom()` заменена функцией `BCryptGenRandom()` из Cryptography API: Next Generation (CNG). PRNG Windows получает энтропию от драйвера режима ядра `cng.sys` (ранее `ksecdd.sys`), сборщик энтропии которого в общих чертах основан на Fortuna. Как это часто бывает в Windows, процесс сложен.

В листинге 2-3 показан типичный вызов CryptGenRandom() на C++ со всеми необходимыми проверками.

```
int random_bytes(unsigned char *out, size_t outlen)
{
    static HCRYPTPROV handle = 0; /* Освобождается только при 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;
}
```

Листинг 2-3. Использование интерфейса PRNG CryptGenRandom() в Windows

Перед вызовом самого PRNG необходимо объявить поставщика криптографических служб (HCRYPTPROV), а затем получить криптографический контекст с помощью CryptAcquireContext(), что повышает вероятность ошибки. Например, было обнаружено, что в последней версии программы шифрования TrueCrypt функция CryptAcquireContext() вызывалась так, что могла незаметно завершиться сбоем, приводя к неоптимальной случайности без уведомления пользователя. К счастью, новый и более простой интерфейс BCryptGenRandom() в Windows не требует от кода явно открывать дескриптор (или, по крайней мере, значительно облегчает использование без дескриптора).

Аппаратный PRNG: Intel Secure Key

До сих пор мы обсуждали только программные PRNG, поэтому теперь рассмотрим аппаратный. Intel Digital Random Number Generator, также называемый Intel Secure Key, - аппаратный PRNG, представленный в 2012 году в микроархитектуре Intel Ivy Bridge. Он основан на рекомендациях NIST SP 800-90 и стандарте Advanced Encryption Standard (AES) в режиме CTR_DRBG. Доступ к PRNG Intel осуществляется через ассемблерную инструкцию RDRAND, которая предоставляет интерфейс, независимый от операционной системы, и в принципе работает быстрее программных PRNG.

В то время как программные PRNG пытаются собирать энтропию из непредсказуемых источников, Intel Secure Key располагает единственным источником энтропии, выдающим последовательный поток данных энтропии в виде нулей и единиц. В терминах аппаратной инженерии этот источник энтропии представляет собой двойную дифференциальную защёлку с обратной связью и режимом конфликтного состояния: по существу, небольшую аппаратную схему, которая переключается между двумя состояниями, 0 и 1, в зависимости от колебаний теплового шума с частотой 3 ГГц. Обычно она довольно надёжна.

Ассемблерная инструкция RDRAND принимает в качестве аргумента регистр размером 16, 32 или 64 бита и записывает в него случайное значение. При вызове RDRAND устанавливает флаг переноса в 1, если данные в регистре назначения являются допустимым случайным значением, и в 0 в противном случае; если вы непосредственно пишете ассемблерный код, обязательно проверяйте флаг CF. Обратите внимание: встроенные функции C, доступные в распространённых компиляторах, не проверяют флаг CF, но возвращают его значение.

> **Примечание.** Платформа PRNG Intel предоставляет помимо `RDRAND` ещё одну ассемблерную инструкцию: `RDSEED` возвращает случайные биты непосредственно из источника энтропии после некоторой обработки, нормализующей или криптографической. Она предназначена для получения начальных значений другими PRNG.

Intel Secure Key документирован лишь частично, однако построен на известных стандартах и прошёл аудит у уважаемой компании Cryptography Research (см. её отчёт «Analysis of Intel's Ivy Bridge Digital Random Number Generator»). Тем не менее высказывались опасения по поводу его безопасности, особенно после разоблачений Эдварда Сноудена о криптографических бэкдорах: PRNG и в самом деле представляет собой идеальную цель для саботажа. Если вас это беспокоит, но вы всё же хотите использовать `RDRAND` или `RDSEED`, смешивайте их выход с другими источниками энтропии. За исключением самых невероятных сценариев, это предотвратит успешную эксплуатацию гипотетического бэкдора в аппаратной части Intel Secure Key или в связанного с ней микрокода.

Что может пойти не так

В заключение я приведу несколько примеров сбоев случайности. Выбирать можно из бесчисленного множества случаев, но я остановился на четырёх, достаточно простых для понимания и иллюстрирующих разные проблемы.

Плохие источники энтропии

В 1996 году реализация SSL в браузере Netscape вычисляла 128-битные начальные значения PRNG согласно псевдокоду из листинга 2-4, скопированному со страницы Голдберга и Вагнера https://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html.

```
глобальная переменная seed;

RNG_CreateContext()
    (seconds, microseconds) = текущее время; /* Время, прошедшее с 1970 года */
    pid = идентификатор процесса;  ppid = идентификатор родительского процесса;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));                          /* 1 */
    seed = MD5(a, b); /* Получение 128-битного значения с помощью хеша MD5 */

mklcpr(x) /* Не имеет криптографического значения; приведено для полноты */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* Очень хорошая стандартная перемешивающая функция; исходный код опущен */
```

Листинг 2-4. Псевдокод создания 128-битных начальных значений PRNG браузером Netscape

Проблема в том, что PID и микросекунды можно угадать. Если предположить, что значение секунд удастся угадать, у микросекунд имеется лишь `10^6` возможных значений, а значит, энтропия `log(10^6)`, или около 20 бит. Идентификатор процесса (PID) и идентификатор родительского процесса (PPID) являются 15-битными значениями, поэтому можно ожидать дополнительных `15 + 15 = 30` бит энтропии. Но если посмотреть, как вычисляется `b` **1**, видно, что перекрытие трёх битов даёт энтропию около `15 + 12 = 27` бит, а общая энтропия составляет лишь 47 бит, хотя у 128-битного начального значения должно быть 128 бит энтропии.

Недостаточная энтропия при загрузке

В 2012 году исследователи просканировали интернет и собрали открытые ключи из сертификатов TLS и узлов SSH. Они обнаружили, что у небольшого числа систем открытые ключи совпадали, а в некоторых случаях были очень похожи (а именно, ключи RSA имели общие простые множители). Иными словами, имелись два числа `n = pq` и `n' = p'q'` при `p = p'`, хотя в норме все значения `p` и `q` в разных модулях должны различаться.

Оказалось, что многие устройства создавали открытый ключ слишком рано, при первой загрузке, прежде чем успевали собрать достаточно энтропии, хотя использовали в остальном приличный PRNG (обычно `/dev/urandom`). Из-за одинакового источника энтропии, например жёстко заданного начального значения, PRNG в разных системах создавали одинаковые случайные биты.

На высоком уровне одинаковые ключи возникают из-за схем создания ключей, подобных следующей, представленной псевдокодом:

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

Если две системы выполнят этот код с одинаковым начальным значением, они создадут одинаковое `p`, одинаковое `q`, а следовательно, и одинаковое `n`.

Общие простые множители в разных ключах возникают в схемах создания ключей, где во время процесса добавляется дополнительная энтропия, как показано 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 начальным значением, они создадут одинаковое `p`, но добавление энтропии посредством `prng.add_entropy()` обеспечит разные значения `q`.

Проблема общих простых множителей состоит в том, что при известных `n = pq` и `n' = pq'` общее значение `p` элементарно восстанавливается вычислением наибольшего общего делителя (НОД) чисел `n` и `n'`. Подробности приведены в статье Хенингер, Дюремерика, Вустроу и Халдермана «Mining Your Ps and Qs», доступной по адресу https://factorable.net.

Некриптографический PRNG

Выше мы обсуждали различие между криптографическими и некриптографическими PRNG и причину, по которой последние никогда нельзя использовать в криптографических приложениях. Увы, многие системы упускают эту деталь, поэтому рассмотрим один такой пример.

Популярное приложение MediaWiki работает на Wikipedia и множестве других вики-сайтов. Случайность используется в нём для создания, помимо прочего, маркеров безопасности и временных паролей, которые должны быть непредсказуемыми. К сожалению, одна ныне устаревшая версия MediaWiki создавала эти маркеры и пароли с помощью некриптографического PRNG Mersenne Twister. Ниже приведён фрагмент уязвимого исходного кода MediaWiki; найдите функцию, вызываемую для получения случайного бита, и прочитайте комментарии:

```
/**
         * Создаёт похожий на шестнадцатеричное число случайный маркер
         * для различных применений.
         * Можно было бы сделать криптографически надёжнее, если это кого-нибудь волнует.
         * @return string
         */
function generateToken( $salt = '' ) {
    $token = dechex(mt_rand()).dechex(mt_rand());
    return md5( $token . $salt );
}
```

Заметили `mt_rand()` в приведённом коде? Здесь `mt` означает Mersenne Twister. В 2012 году исследователи показали, как воспользоваться предсказуемостью Mersenne Twister, чтобы по паре маркеров безопасности предсказывать будущие маркеры и временные пароли. MediaWiki исправили, переведя на криптографический PRNG.

Ошибка выборки при сильной случайности

Следующая ошибка показывает, как даже сильный криптографический PRNG с достаточной энтропией может создавать смещённое распределение. Программа чата Cryptocat была предназначена для защищённой связи. В ней использовалась функция, пытавшаяся создать равномерно распределённую строку десятичных цифр, то есть чисел от 0 до 9. Однако простое вычисление случайных байтов по модулю 10 не даёт равномерного распределения: если взять все числа от 0 до 255 и привести их по модулю 10, количество получившихся значений от 0 до 9 будет неодинаковым.

Чтобы решить эту проблему и получить равномерное распределение, Cryptocat выполняла следующее:

```
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)
}
```

И это было почти безупречно. Если брать только числа до значения, кратного 10, а остальные отбрасывать, можно ожидать равномерного распределения цифр от 0 до 9. К сожалению, в условии `if` была ошибка на единицу. Подробности я оставлю вам в качестве упражнения. Вы должны обнаружить небольшое статистическое смещение в пользу индекса 0 (подсказка: вместо `<=` следовало использовать `<`).

Дополнительная литература

Я лишь слегка коснулся темы случайности в криптографии. О теории случайности можно узнать гораздо больше: существуют различные понятия энтропии, экстракторы случайности и даже вопросы о возможностях рандомизации и дерандомизации в теории сложности. Чтобы подробнее узнать о PRNG и их безопасности, прочитайте классическую статью 1998 года Келси, Шнайера, Вагнера и Холла «Cryptanalytic Attacks on Pseudorandom Number Generators». Затем изучите реализации PRNG в своих любимых приложениях и попробуйте найти их слабости. (Поищите в интернете «random generator bug» - и найдёте множество примеров.)

Но со случайностью мы ещё не закончили. Она встретится в этой книге не раз, и вы узнаете множество способов, которыми она помогает строить безопасные системы.

Глава 3. Криптографическая безопасность

Криптографические определения безопасности отличаются от тех, что применяются к компьютерной безопасности в целом. Главное различие между безопасностью программного обеспечения и криптографической безопасностью состоит в том, что последнюю можно измерить количественно. В мире программного обеспечения мы обычно говорим, что приложения либо безопасны, либо небезопасны, а в мире криптографии зачастую можно вычислить объём усилий, необходимый для взлома криптографического алгоритма. Кроме того, безопасность программного обеспечения сосредоточена на том, чтобы не позволить злоумышленникам воспользоваться кодом программы во вред, тогда как цель криптографической безопасности - сделать строго определённые задачи неразрешимыми.

Криптографические задачи связаны с математическими понятиями, но не со сложной математикой - во всяком случае, не в этой книге. В данной главе рассматриваются некоторые из этих понятий безопасности и способы их применения к решению реальных задач.

В следующих разделах я расскажу, как количественно измерять криптографическую безопасность способами, одновременно корректными в теории и значимыми на практике. Мы обсудим безусловную и вычислительную безопасность, битовую стойкость и полную стоимость атаки, доказуемую и эвристическую безопасность, а также создание симметричных и асимметричных ключей. В заключение главы будут приведены реальные примеры отказов в криптографии, которая казалась стойкой.

Определение невозможного

В главе 1 я описал безопасность шифра относительно возможностей и целей злоумышленника и назвал шифр безопасным, если при известных возможностях злоумышленник не может достичь этих целей. Но что в данном контексте означает «не может»?

В криптографии понятие невозможности определяется двумя видами безопасности: безусловной и вычислительной. Грубо говоря, безусловная безопасность означает теоретическую невозможность, а вычислительная - практическую. Безусловная безопасность не измеряет стойкость количественно, потому что рассматривает шифр либо как безопасный, либо как небезопасный, не допуская промежуточных состояний; поэтому на практике она бесполезна, хотя играет важную роль в теоретической криптографии. Вычислительная безопасность - более уместная и практичная мера стойкости шифра.

Безопасность в теории: безусловная безопасность

Безусловная безопасность основывается не на том, насколько трудно взломать шифр, а на том, мыслим ли его взлом вообще. Шифр безусловно безопасен только в том случае, если его нельзя взломать даже при неограниченном времени вычислений и объёме памяти. Если успешная атака на шифр заняла бы триллионы лет, такой шифр всё равно безусловно небезопасен.

Например, одноразовый блокнот из главы 1 безусловно безопасен. Вспомним: одноразовый блокнот шифрует открытый текст `P` в шифротекст `C = P xor K`, где `K` - случайная битовая строка, уникальная для каждого открытого текста. Шифр безусловно безопасен, поскольку, имея шифротекст и неограниченное время на перебор всех возможных ключей `K` и вычисление соответствующих открытых текстов `P`, вы всё равно не сможете определить правильный `K`:

возможных значений P столько же, сколько значений K.

Безопасность на практике: вычислительная безопасность

В отличие от безусловной, вычислительная безопасность считает шифр безопасным, если его невозможно взломать за разумное время и с разумными ресурсами, такими как память, оборудование, бюджет и энергия. Вычислительная безопасность позволяет количественно измерить стойкость шифра или любого другого криптографического алгоритма.

Например, рассмотрим шифр E, для которого известна пара «открытый текст - шифротекст» (P, C), но неизвестен 128-битный ключ K, вычисляющий C = E(K, P). Этот шифр не является безусловно безопасным, поскольку его можно взломать, перебрав 2^128 возможных 128-битных значений K, пока не найдётся удовлетворяющее E(K, P) = C. Но на практике, даже проверяя 100 миллиардов ключей в секунду, вы потратите более 100 000 000 000 000 000 000 лет. Иными словами, с разумной точки зрения такой шифр вычислительно безопасен, поскольку его практически невозможно взломать.

Вычислительную безопасность можно выразить двумя величинами:

• t - предел числа операций, которые выполнит злоумышленник;

• epsilon (эпсилон) - верхняя граница вероятности успеха атаки.

Мы говорим, что криптографическая схема (t, epsilon)-безопасна, если вероятность успеха злоумышленника, выполняющего не более t операций, какими бы эти операции ни были, не превышает epsilon, где epsilon лежит в диапазоне от 0 до 1 включительно. Вычислительная безопасность задаёт предел сложности взлома криптографического алгоритма.

Важно понимать, что t и epsilon - лишь границы: если шифр (t, epsilon)-безопасен, ни один злоумышленник, выполняющий меньше t операций, не добьётся успеха с вероятностью epsilon. Однако отсюда не следует, что злоумышленник, выполнивший ровно t операций, добьётся успеха; это также не сообщает необходимого числа операций, которое может быть значительно больше t. Мы говорим, что t - нижняя граница необходимых вычислительных усилий, поскольку для нарушения безопасности понадобится как минимум t операций.

Если точно известно, сколько усилий требуется для взлома шифра, говорят, что (t, epsilon)-безопасность задаёт точную границу, когда существует атака, взламывающая шифр с вероятностью epsilon ровно за t операций.

Например, рассмотрим симметричный шифр со 128-битным ключом. В идеале он должен быть (t, t/2^128)-безопасным для любого t от 1 до 2^128. Лучшей атакой должен оставаться полный перебор, то есть проверка всех ключей до обнаружения правильного. Любая более эффективная атака должна использовать какой-либо изъян шифра, поэтому мы стремимся создавать шифры, против которых полный перебор является наилучшей возможной атакой.

Рассмотрим утверждение о (t, t/2^128)-безопасности и вероятность успеха трёх возможных атак:

• В первом случае t = 1: злоумышленник проверяет один ключ и добивается успеха с вероятностью epsilon = 1/2^128.

• Во втором случае t = 2^128: злоумышленник проверяет все 2^128 ключей, и один из них подходит. Следовательно, вероятность epsilon = 1. (Если злоумышленник перебирает все ключи, правильный непременно окажется среди них.)

• В третьем случае злоумышленник проверяет лишь t = 2^64 ключей и добивается успеха с вероятностью epsilon = 2^64/2^128 = 2^(-64). Когда злоумышленник проверяет лишь часть

всех ключей, вероятность успеха пропорциональна числу проверенных ключей.

Можно заключить, что шифр с n-битным ключом в лучшем случае является (t, t/2^n)-безопасным для любого t от 1 до 2^n, поскольку атака полным перебором всегда добьётся успеха независимо от стойкости шифра. Поэтому ключ должен быть достаточно длинным, чтобы на практике сделать атаки полным перебором бесполезными.

> **Примечание.** В этом примере подсчитывается число вычислений шифра, а не абсолютное время или число тактов процессора. Вычислительная безопасность не зависит от технологии: шифр, являющийся (t, epsilon)-безопасным сегодня, останется (t, epsilon)-безопасным и завтра, но то, что считается практически безопасным сегодня, завтра может уже таковым не считаться.

Количественная оценка безопасности

Обнаружив атаку, сначала следует определить её теоретическую эффективность и, если она вообще осуществима, практичность. Аналогично, имея шифр, объявленный безопасным, вы захотите узнать, какой объём работы он способен выдержать. Чтобы ответить на эти вопросы, я объясню, как измерять криптографическую стойкость в битах (теоретический взгляд) и какие факторы влияют на реальную стоимость атаки.

Измерение стойкости в битах

Говоря о вычислительной безопасности, шифр называют t-стойким, если успешная атака требует как минимум t операций. Тем самым мы избегаем неинтуитивной записи (t, epsilon), предполагая вероятность успеха epsilon, близкую к 1, либо любую другую вероятность, значимую для нас на практике. Затем стойкость выражают в битах: «n-битная стойкость» означает, что для нарушения некоторого конкретного понятия безопасности требуется около 2^n операций.

Если приблизительно известно, сколько операций требуется для взлома шифра, уровень его стойкости в битах можно определить, взяв двоичный логарифм числа операций: если требуется 1 000 000 операций, уровень стойкости равен log_2(1 000 000), то есть примерно 20 битам, поскольку 1 000 000 ~= 2^20. Вспомним, что n-битный ключ обеспечивает не более n бит стойкости, поскольку атака полным перебором всех 2^n возможных ключей всегда добьётся успеха. Но размер ключа не всегда совпадает с уровнем стойкости: он задаёт лишь верхнюю границу, то есть максимально возможный уровень.

Уровень стойкости может быть ниже размера ключа по одной из двух причин:

• Атака взломала шифр за меньшее, чем ожидалось, число операций, например методом, восстанавливающим ключ посредством проверки лишь части из 2^n ключей.

• Уровень стойкости шифра намеренно отличается от размера ключа, как у большинства алгоритмов с открытым ключом. Например, алгоритм RSA с 1024-битными элементами закрытого ключа и, соответственно, 2048-битным модулем обеспечивает менее 128 бит стойкости.

Битовая стойкость полезна при сравнении уровней стойкости шифров, но не даёт достаточно информации о фактической стоимости атаки. Иногда это слишком простая абстракция, поскольку предполагается, что для взлома шифра с n-битной стойкостью требуется 2^n операций независимо от того, что это за операции. Поэтому два шифра с одинаковым уровнем битовой стойкости могут обладать совершенно разной стойкостью в реальном мире, если учесть фактическую стоимость

атаки для настоящего злоумышленника.

Предположим, имеются два шифра, каждый со 128-битным ключом и 128-битной стойкостью. Для взлома каждого шифр нужно вычислить 2^128 раз, но второй шифр в 100 раз медленнее первого. Поэтому 2^128 вычислений второго шифра занимают столько же времени, сколько 100 * 2^128 ~= 2^(134,64) вычислений первого. Если вести подсчёт в единицах первого, быстрого шифра, взлом медленного потребует 2^(134,64) операций. Если считать в единицах второго, медленного шифра, потребуется лишь 2^128 операций. Следует ли тогда считать второй шифр сильнее первого? В принципе да, но среди распространённых шифров стократное различие производительности встречается редко.

Непоследовательное определение операции создаёт дополнительные трудности при сравнении эффективности атак. В некоторых атаках заявляется снижение стойкости шифра, поскольку они выполняют 2^120 вычислений некоторой операции вместо 2^128 вычислений шифра, но скорость каждого типа атаки в анализе не учитывается. Атака из 2^120 операций не всегда окажется быстрее полного перебора из 2^128 операций.

Тем не менее битовая стойкость остаётся полезным понятием, если операция определена разумно, то есть выполняется примерно с той же скоростью, что и вычисление шифра. В конце концов, в реальной жизни, чтобы определить достаточность уровня стойкости, требуется лишь порядок величины.

Расчёт полной стоимости атаки

Битовая стойкость выражает стоимость самой быстрой атаки на шифр, оценивая порядок величины необходимого для успеха числа операций. Но на стоимость атаки влияют и другие факторы, которые необходимо учитывать при оценке фактического уровня стойкости. Я объясню четыре основных: параллелизм, память, предварительные вычисления и число целей.

Параллелизм

Первый фактор - вычислительный параллелизм, то есть способность реализации атаки использовать параллельные вычисления, например многоядерные системы.

Для примера рассмотрим две атаки по 2^56 операций каждая:

• Первая атака выполняет 2^56 последовательно зависимых операций, вычисляя x_(i+1) = f_i(x_i) для некоторого фиксированного x_0 и различных функций f_i, где i изменяется от 1 до 2^56.

• Вторая атака выполняет 2^56 независимых операций, вычисляя x_i = f_i(x) для некоторого фиксированного x и различных функций f_i, где i изменяется от 1 до 2^56. Вычисления f_i(x) можно выполнять параллельно, поскольку каждое из них не зависит от остальных.

Различие между двумя атаками состоит в том, что вторую можно распараллелить, а первую - нет. Параллельная обработка может быть на порядки быстрее последовательной. Например, если в вашем распоряжении 2^16 = 65 536 процессоров, рабочую нагрузку параллельной атаки можно разделить на 2^16 независимых задач, каждая из которых выполняет 2^56/2^16 = 2^40 операций. Однако первая атака не сможет воспользоваться наличием нескольких ядер поскольку каждая операция зависит от результата предыдущей. Поэтому параллельная атака завершится в 65 536 раз быстрее последовательной, хотя обе выполняют одинаковое число операций.

> **Примечание.** Алгоритмы, атака на которые при наличии N ядер ускоряется в N раз, называют идеально параллельными; время их выполнения линейно масштабируется относительно числа вычислительных ядер.

Память

Второй фактор, определяющий стоимость атаки, - память. Криптоаналитические атаки оцениваются с точки зрения использования времени и пространства: сколько операций они выполняют за некоторое время, сколько памяти, или пространства, потребляют, как используют это пространство и какова скорость доступной памяти? К сожалению, битовая стойкость учитывает лишь время выполнения атаки.

Что касается использования пространства, важно учитывать число необходимых атаке обращений к памяти, скорость доступа к памяти (которая может различаться для чтения и записи), размер получаемых данных, схему доступа (последовательные или случайные адреса памяти) и способ организации данных в памяти. Например, на процессоре Intel Xeon 8380 Ice Lake 2021 года обращение к регистру занимает один такт, к кешу L1 (48 Кбайт) - 5 тактов, к кешу L2 (1,25 Мбайт) - 14 тактов, к кешу L3 (60 Мбайт) - 63,5 такта, а обращение к DRAM в лучшем случае столь же быстро, но зачастую значительно медленнее обращения к кешу L3 (точная задержка зависит от нескольких факторов).

Предварительные вычисления

Операции предварительного вычисления требуется выполнить лишь однажды, после чего их результаты можно повторно использовать при последующих выполнениях атаки. Иногда предварительные вычисления называют автономным этапом атаки.

Рассмотрим атаку с компромиссом «время - память», при которой злоумышленник выполняет одно огромное вычисление, создающее большие таблицы поиска, а затем сохраняет и многократно использует эти таблицы в самой атаке. Например, в одной атаке на шифрование мобильной связи 2G построение таблиц объёмом 2 Тбайт заняло два месяца, после чего злоумышленники могли с их помощью взломать шифрование 2G и восстановить секретный сеансовый ключ всего за несколько секунд.

Число целей

Наконец, рассмотрим число целей атаки. Чем больше целей, тем шире поверхность атаки и тем больше злоумышленники могут узнать об искомых ключах.

В качестве примера рассмотрим полный перебор ключей. Если целью является единственный n-битный ключ, для гарантированного нахождения правильного ключа потребуется 2^n попыток. Если целями являются несколько n-битных ключей, скажем M ключей, и для единственного P имеются M различных шифротекстов, где C = E(K, P) для каждого из искомых M ключей K, то для нахождения каждого ключа снова потребуется 2^n попыток. Но если вас интересует хотя бы один из M ключей, а не все сразу, для успеха в среднем понадобится 2^n/M попыток. Например, чтобы взломать один 128-битный ключ из 2^16 = 65 536 целевых ключей, в среднем потребуется 2^(128-16) = 2^112 вычислений шифра. Иными словами, с ростом числа целей стоимость атаки снижается, а её скорость возрастает.

Выбор и оценка уровней стойкости

Выбор уровня стойкости зачастую означает выбор между 128-битным и 256-битным уровнями, доступными в большинстве стандартных криптографических алгоритмов и реализаций. Можно встретить схемы с 64- или 80-битной стойкостью, но в целом для реального применения они недостаточно безопасны.

На высоком уровне 128-битная стойкость означает, что для взлома криптосистемы понадобится выполнить приблизительно 2^128 операций. Чтобы вы могли ощутить величину этого числа, учтите: возраст Вселенной составляет приблизительно 2^88 наносекунд (в секунде миллиард наносекунд). Поскольку при современной технологии проверка ключа занимает не меньше наносекунды, для успешной атаки понадобится время, в несколько раз превышающее возраст Вселенной, а точнее в 2^40 раз.

Но разве параллелизм и множество целей не способны радикально сократить время успешной атаки? Не совсем. Предположим, вы хотите взломать любую из миллиона целей и располагаете миллионом параллельных ядер. Тогда время поиска уменьшается с 2^128 до (2^128/2^20)/2^20 = 2^88, что эквивалентно «всего лишь» одному возрасту Вселенной.

При оценке уровней стойкости следует учитывать и развитие технологий. Закон Мура утверждает, что эффективность вычислений приблизительно удваивается каждые два года. Это можно рассматривать как потерю одного бита стойкости каждые два года: если сегодня бюджет в 1000 долларов позволяет взломать, скажем, 40-битный ключ за один час, то, согласно закону Мура, через два года за тот же бюджет в 1000 долларов можно будет за час взломать 41-битный ключ (я упрощаю). Экстраполируя, можно сказать, что, согласно закону Мура, через 80 лет у нас будет на 40 бит стойкости меньше, чем сегодня. Иными словами, через 80 лет выполнение 2^128 операций может стоить столько же, сколько сегодня выполнение 2^88 операций. С учётом параллелизма и множества целей мы опускаемся до 2^48 наносекунд вычислений, то есть примерно трёх дней. Но эта экстраполяция крайне неточна, поскольку закон Мура не будет и не может масштабироваться настолько долго. И всё же общий смысл понятен: то, что сегодня кажется неосуществимым, через столетие может стать реальным.

Иногда уровень стойкости ниже 128 бит оправдан, например если безопасность нужна лишь на короткий срок, а реализация более высокого уровня отрицательно скажется на стоимости или удобстве системы. Примером служат системы платного телевидения, в которых ключи шифрования имеют длину 48 или 64 бита. Это кажется до смешного малым, но вполне достаточно, поскольку ключ обновляется каждые 5 или 10 секунд.

Тем не менее для долговременной безопасности следует выбирать 256-битную или чуть меньшую стойкость. Даже при худшем сценарии - существовании квантовых компьютеров (см. главу 14) - в обозримом будущем схему с 256-битной стойкостью вряд ли удастся взломать. Стойкость выше 256 бит практически не нужна, разве что как маркетинговый приём.

Как однажды выразился криптограф Джон Келси: «Разница между перебором 80-битных и 128-битных ключей подобна разнице между полётом на Марс и полётом к Альфе Центавра. Насколько я могу судить, с точки зрения практического полного перебора между 192- и 256-битными ключами нет содержательной разницы; невозможное остаётся невозможным».

Достижение безопасности

Выбрав уровень стойкости, важно гарантировать, что криптографические схемы будут ему соответствовать. Иными словами, вам нужна уверенность, а не просто надежда и неизвестность, что всё всегда будет работать согласно плану.

При формировании уверенности в безопасности криптографического алгоритма можно опираться на математические доказательства - подход, называемый доказуемой безопасностью, - либо на свидетельства неудачных попыток взломать алгоритм, которые я буду называть эвристической безопасностью (иногда её называют вероятной безопасностью). Эти два подхода дополняют друг друга, и ни один из них не лучше другого, как вы увидите далее.

Доказуемая безопасность

Доказуемая безопасность означает доказательство того, что взлом криптографической схемы не легче решения другой задачи, известной своей трудностью. Такое доказательство безопасности гарантирует, что криптография останется безопасной до тех пор, пока трудная задача остаётся трудной. Доказательство этого типа называется редукцией и происходит из теории сложности. Говорят, что задача X сводится к взлому некоторого шифра, если любой метод взлома шифра даёт также метод решения задачи X. Такая редукция гарантирует: пока задача X трудна, шифр безопасен.

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

Доказательства относительно математической задачи

Многие доказательства безопасности, например для криптографии с открытым ключом, показывают, что взлом криптографической схемы не легче решения некоторой трудной математической задачи. Речь идёт о задачах, для которых известно, что решение существует и легко проверяется, когда оно известно, но найти его вычислительно трудно.

> **Примечание.** Настоящего доказательства того, что кажущиеся трудными математические задачи действительно трудны, не существует. Фактически доказать это для определённого класса задач - одна из величайших проблем теории сложности. На момент написания книги Математический институт Клэя назначил награду в 1 000 000 долларов тому, кто сумеет это доказать. Подробнее я расскажу об этом в главе 9.

Рассмотрим, например, задачу факторизации - самую известную математическую задачу в криптографии: по числу, которое, как известно, является произведением двух простых чисел (`n = pq`), найти эти простые числа. Например, если `n = 15`, ответом будут 3 и 5. Для малого числа это просто, но с ростом размера числа задача экспоненциально усложняется. Считается, что факторизация практически неосуществима, если число `n` имеет длину 3000 бит (около 900 десятичных цифр) или более.

Rivest-Shamir-Adleman (RSA) - самая известная криптографическая схема, опирающаяся на задачу факторизации. RSA шифрует открытый текст `P`, рассматриваемый как большое число, вычисляя `C = P^e mod n`, где числа `e` и `n = pq` образуют открытый ключ. Расшифрование восстанавливает открытый текст из шифротекста вычислением `P = C^d mod n`, где `d` - закрытый ключ, связанный с `e` и `n`. Если мы можем факторизовать `n`, то можем взломать RSA, восстановив закрытый ключ по

открытому; а если можем получить закрытый ключ, то можем факторизовать n (см., например, статью https://eprint.iacr.org/2004/208). Иными словами, восстановление закрытого ключа RSA и факторизация n - задачи равной трудности. Именно такая редукция нужна в доказуемой безопасности. Однако нет гарантии, что восстановить открытый текст RSA так же трудно, как факторизовать n, поскольку знание открытого текста не раскрывает закрытый ключ.

Доказательства относительно другой криптографической задачи

Вместо сравнения криптографической схемы с математической задачей можно сравнить её с другой криптографической схемой и доказать, что вторую схему удастся взломать только при возможности взломать первую. Доказательства безопасности симметричных шифров обычно следуют этому подходу.

Например, имея лишь один алгоритм перестановки, можно построить симметричные шифры, генераторы случайных битов и другие криптографические объекты, такие как хеш-функции, сочетая вызовы перестановки с различными типами входных данных (как вы увидите в главе 6). Затем доказательства показывают, что вновь созданные схемы безопасны, если безопасна перестановка. Иными словами, мы знаем, что новый алгоритм не слабее исходного. Такие доказательства обычно строятся путём создания атаки на меньший компонент из атаки на больший, то есть демонстрацией редукции.

Когда доказывается, что криптографический алгоритм не слабее другого, главное преимущество заключается в уменьшении поверхности атаки: вместо анализа и базового алгоритма, и их сочетания достаточно рассмотреть базовый алгоритм нового шифра. В частности, если вы пишете шифр, использующий новую перестановку и новый способ комбинирования, можно доказать, что сочетание не ослабляет безопасность по сравнению с базовым алгоритмом. Следовательно, для взлома сочетания потребуется взломать новую перестановку.

Оговорки

Исследователи криптографии в значительной степени полагаются на доказательства безопасности как относительно математических задач, так и относительно других криптографических схем. Но наличие доказательства безопасности не гарантирует совершенства криптографической схемы и не оправдывает пренебрежение более практическими аспектами реализации. В конце концов, как однажды сказал криптограф Ларс Кнудсен: «Если безопасность доказана, вероятно, её нет», имея в виду, что доказательство безопасности нельзя принимать за абсолютную гарантию. Хуже того, «доказуемо безопасная» схема может привести к нарушению безопасности по нескольким причинам.

Одна из проблем связана с самим выражением «доказательство безопасности». В математике доказательство демонстрирует абсолютную истину, но в криптографии - лишь относительную. Например, доказательство того, что взлом вашего шифра столь же труден, как вычисление дискретных логарифмов, то есть нахождение числа `x` по `g` и `g^x mod n`, гарантирует: если ваш шифр окажется несостоятелен, вместе с ним окажется несостоятельным огромное множество других шифров, и в случае худшего никто не станет винить вас.

Ещё одна оговорка: обычно безопасность доказывают относительно единственного понятия безопасности. Например, можно доказать, что восстановить закрытый ключ шифра столь же трудно, как решить задачу факторизации. Но если открытые тексты можно восстанавливать из шифротекстов без ключа, вы обойдёте доказательство, и восстановление ключа будет едва ли важно.

Кроме того, доказательства не всегда верны, и взломать алгоритм может оказаться легче, чем считалось изначально.

> **Примечание.** К сожалению, лишь немногие исследователи тщательно проверяют доказательства безопасности, которые нередко занимают десятки страниц, что затрудняет контроль качества. При этом демонстрация неверности доказательства не обязательно означает, что доказываемое утверждение полностью ошибочно; если результат верен, доказательство иногда удаётся спасти, исправив ошибки.

Ещё одно важное соображение: трудные математические задачи порой оказываются легче, чем ожидалось. Например, некоторые слабые параметры упрощают взлом криптосистемы RSA. Или математическая задача может быть трудной в отдельных случаях, но не в среднем, как часто происходит, когда эталонная задача нова и плохо изучена. Именно это случилось со схемой шифрования «рюкзак» Меркла и Хеллмана 1978 года, которую позднее взломали методами редукции решёток.

Наконец, доказательство безопасности алгоритма может быть безупречным, а реализация алгоритма - слабой. Например, злоумышленники могут использовать сведения из побочных каналов, такие как энергопотребление или время выполнения, чтобы узнать о внутренних операциях алгоритма и взломать его, обойдя доказательство. Или разработчики могут неправильно применить криптографическую схему: чем сложнее алгоритм и чем больше в нём настраиваемых параметров, тем выше вероятность, что пользователь или разработчик ошибётся в конфигурации, сделав алгоритм полностью небезопасным.

Эвристическая безопасность

Доказуемая безопасность - прекрасный инструмент для обретения уверенности в криптографической схеме, но применим он не ко всем типам алгоритмов. Фактически для большинства симметричных шифров доказательства безопасности не существует. Например, каждый день мы полагаемся на AES для защищённой связи с помощью мобильных телефонов, ноутбуков и настольных компьютеров, однако безопасность AES не доказана: не существует доказательства, что взломать его столь же трудно, как решить некоторую известную задачу. AES нельзя связать с математической задачей или другим алгоритмом, потому что он сам и является трудной задачей.

Когда доказуемая безопасность неприменима, единственная причина доверять шифру состоит в том, что множество квалифицированных специалистов пытались его взломать и потерпели неудачу. Это и называется эвристической безопасностью.

Когда можно быть уверенным, что шифр безопасен? Абсолютная уверенность невозможна, но можно с большой уверенностью считать, что алгоритм не будет взломан, если сотни опытных криптоаналитиков потратили по сотне часов каждый на попытки взлома и опубликовали результаты. Обычно они пытаются атаковать упрощённые версии шифра, часто с меньшим числом операций или раундов - коротких последовательностей операций, которые шифр повторяет, чтобы перемешивать биты.

Анализируя новый шифр, криптоаналитики сначала пытаются взломать один раунд, затем два, три и столько, сколько смогут. Запас стойкости - это разность между общим числом раундов и числом успешно атакованных раундов. Если после многолетнего изучения запас стойкости шифра остаётся большим, мы обретаем уверенность, что он (вероятно) безопасен.

Создание ключей

Если вы собираетесь что-либо шифровать, придётся создавать ключи: временные «сеансовые ключи», подобные создаваемым при посещении сайта HTTPS, или долговременные открытые ключи. Вспомним из главы 2, что секретные ключи лежат в основе криптографической безопасности и должны создаваться случайно, чтобы оставаться непредсказуемыми и секретными.

Например, когда вы посещаете сайт HTTPS, браузер получает открытый ключ сайта и использует его для установления симметричного ключа, действующего лишь в текущем сеансе; при этом открытый ключ сайта и связанный с ним закрытый ключ могут оставаться действительными годами. Поэтому злоумышленнику должно быть трудно их найти. Но создание секретного ключа не всегда сводится к выдаче достаточного количества псевдослучайных битов. Криптографические ключи можно создавать одним из трёх способов:

• Случайно, с помощью PRNG, подающего данные алгоритму создания ключей.

• Из пароля, с помощью основанной на пароле функции выработки ключа (password-based key derivation function, PBKDF), преобразующей предоставленный пользователем пароль в ключ.

• Посредством протокола согласования ключей - последовательности сообщений между двумя или несколькими сторонами, завершающейся установлением общего ключа.

Пока я объясню простейший метод: случайное создание.

Симметричные ключи

Симметричные ключи - это секретные ключи, совместно используемые двумя сторонами; создавать их проще всего. Обычно их длина совпадает с обеспечиваемым уровнем стойкости: 128-битный ключ даёт 128-битную стойкость, причём допустим любой из 2^128 возможных ключей.

Чтобы создать n-битный симметричный ключ с помощью криптографического PRNG, достаточно запросить у него n псевдослучайных битов и использовать эти биты как ключ.

Вот и всё. Например, с помощью набора OpenSSL можно создать случайный симметричный ключ, выдав псевдослучайные байты следующей командой:

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

Разумеется, ваш результат будет отличаться от моего.

Асимметричные ключи

В отличие от симметричных, асимметричные ключи обычно длиннее обеспечиваемого ими уровня стойкости. Но главная проблема состоит в том, что создавать асимметричные ключи труднее, чем симметричные: нельзя просто получить от PRNG n битов и удовлетвориться результатом. Асимметричные ключи - не просто необработанные битовые последовательности. Они представляют объекты определённого типа, например большое число с особыми свойствами (в RSA - произведение двух простых чисел). Случайная битовая строка, а значит, и случайное число, вряд ли будет обладать необходимыми свойствами и потому не станет допустимым ключом.

Для создания асимметричного ключа псевдослучайные биты передаются в качестве начального значения алгоритму создания ключей. Этот алгоритм получает начальное значение длиной не меньше требуемого уровня стойкости и строит из него закрытый ключ и соответствующий открытый ключ, гарантируя, что оба удовлетворяют необходимым критериям. Например, наивный алгоритм

создания ключей RSA формировал бы число `n = pq` с помощью алгоритма, создающего два случайных простых числа приблизительно одинаковой длины. Такой алгоритм выбирал бы случайные числа, пока одно из них не окажется простым, поэтому понадобился бы ещё и алгоритм проверки числа на простоту.

Чтобы избавить себя от необходимости вручную реализовывать алгоритм создания ключей, можно воспользоваться OpenSSL и создать 4096-битный закрытый ключ RSA следующим образом:

```
$ openssl genrsa 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.....................................................................
.............................++
.........................................++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MIIJKQIBAAKCAgEA3Qgm6OjMy61YVstaGawk22A9LyMXhiQUU4N8F5QZXEef2Pjq
vTtAIA1hzpK2AJsv16INpNkYcTjNmechAJ0xHraftO6cp2pZFP85dvknsMfUoe8u
btKXZiYvJwpS0fQQ4tzlDtH45Gj8sMHcwFxTO3HSIx0XV0owfJTLMzZbSE3TDlN+
JdW8d9Xd5UVB+o9gUCI8tSfnOjF2dHlLNiOhlfT4w0Rf+G35USIyUJZtOQ0Dh8M+
--snip--
zO/dbYtqRkMT8Ubb/0Q1IW0q8e0WnFetzkwPzAIjwZGXT0kWJu3RYj1OXbTYDr2c
xBRVC/ujoDL6O3NaqPxkWY5HJVmkyKIE5pC04RFNyaQ8+o4APyobabPMylQq5Vo5
N5L2c4mhy1/OH8fvKBRDuvCk2oZinjdoKUo8ZA5DOa4pdvIQfR+b4/4Jjsx4
-----END RSA PRIVATE KEY-----
```

Обратите внимание: ключ имеет определённый формат, а именно данные в кодировке base64 между маркерами `BEGIN RSA PRIVATE KEY` и `END RSA PRIVATE KEY`. Это стандартный формат кодирования, который поддерживается большинством систем и может быть преобразован в необработанные байты данных. Последовательности точек в начале служат своеобразным индикатором выполнения, а строка `e is 65537 (0x10001)` указывает параметр, используемый при шифровании (помните, что RSA шифрует вычислением `C = P^e mod n`).

Защита ключей

Получив секретный ключ, необходимо сохранить его в тайне, но обеспечить доступность в нужный момент. Эту задачу можно решить тремя способами:

• **Обёртывание ключа (шифрование ключа вторым ключом).**

Проблема такого подхода состоит в том, что второй ключ должен быть доступен, когда потребуется расшифровать защищённый ключ. На практике этот второй ключ часто создаётся из пароля, который пользователь вводит, когда ему нужно применить защищённый ключ. Обычно именно так защищают закрытые ключи протокола Secure Shell (SSH).

• **Создание на лету из пароля.**

Этот способ не требует хранения зашифрованного файла, поскольку ключ получается непосредственно из пароля. Такой метод часто применяют системы вроде криптовалютных кошельков и диспетчеров паролей.

Хотя этот метод прямее обёртывания ключа, он распространён меньше, в частности потому, что уязвимее для слабых паролей. Предположим, например, что злоумышленник перехватил зашифрованное сообщение. Если применялось обёртывание ключа, злоумышленнику сначала придётся получить файл защищённого ключа, который может храниться локально в файловой системе пользователя или в системе управления ключами (key management system, KMS) и потому быть труднодоступным. Но если ключ создавался на лету, злоумышленник может напрямую искать правильный пароль, пытаясь расшифровать сообщение паролями-кандидатами. Если пароль слаб, ключ будет скомпрометирован.

• **Хранение ключа в аппаратном токене (смарт-карте или USB-ключе).**

При этом подходе ключ хранится в защищённой памяти и остаётся безопасным, даже если компьютер скомпрометирован. Это самый безопасный способ хранения ключа, но также самый дорогой и наименее удобный, поскольку аппаратный токен приходится носить с собой, рискуя его потерять. Для разблокирования ключа в защищённой памяти смарт-карты и USB-ключи обычно требуют ввода пароля.

> **Примечание.** Какой бы метод вы ни использовали, при обмене ключами не перепутайте закрытый ключ с открытым и случайно не опубликуйте закрытый ключ в электронном письме или исходном коде. (Я действительно находил закрытые ключи на GitHub.)

Чтобы проверить обёртывание ключа, выполните следующую команду OpenSSL с аргументом `-aes128`, указывающим OpenSSL зашифровать ключ шифром AES-128 (AES со 128-битным ключом):

```
$ openssl genrsa -aes128 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
.........++
.....................................................................................
.........................................++
e is 65537 (0x10001)
Enter pass phrase:
```

OpenSSL зашифрует вновь созданный ключ с помощью запрошенной парольной фразы.

Что может пойти не так

Криптографическая безопасность может нарушиться множеством способов. Самый большой риск возникает при ложном чувстве защищённости, порождённом доказательствами безопасности или хорошо изученными протоколами, как показывают следующие примеры.

Неверное доказательство безопасности

Даже доказательства безопасности, предложенные знаменитыми исследователями, могут оказаться ошибочными. Один из самых ярких примеров катастрофически неверного доказательства связан с Optimal Asymmetric Encryption Padding (OAEP) - методом безопасного шифрования на основе RSA, реализованным во многих приложениях. Неверное доказательство стойкости OAEP против атак с выбранным шифротекстом считалось правильным семь лет, пока в 2001 году исследователь не обнаружил изъян. Ошибочным оказалось не только доказательство, но и сам результат. Новое доказательство позднее показало, что OAEP лишь почти безопасен против атак с выбранным шифротекстом. Теперь нам остаётся доверять новому доказательству и надеяться, что оно безупречно. (Дополнительные сведения см. в статье Виктора Шупа 2001 года «OAEP Reconsidered».)

Короткие ключи ради поддержки устаревших систем

В 2015 году исследователи обнаружили, что некоторые сайты HTTPS и серверы SSH поддерживали криптографию с открытым ключом с более короткими ключами, чем ожидалось, а именно 512 бит вместо как минимум 2048 бит. Помните: в схемах с открытым ключом уровень стойкости не равен размеру ключа, и для HTTPS 512-битные ключи обеспечивают уровень стойкости приблизительно 60 бит. Такие ключи можно было взломать всего примерно за две недели вычислений на кластере из 72 процессоров. Проблема затронула множество сайтов, включая сайт Федерального бюро расследований США (FBI). Хотя программное обеспечение в конечном итоге исправили благодаря обновлениям OpenSSL и других программ, эта проблема стала весьма неприятным сюрпризом.

Дополнительная литература

Чтобы больше узнать о доказуемой безопасности симметричных шифров, прочитайте документацию по губчатым функциям https://keccak.team/sponge_duplex.html. Губчатые функции ввели основанный на перестановках подход в симметричной криптографии, описывающий построение множества различных криптографических функций с помощью всего одной перестановки.

К числу обязательных работ о реальной стоимости атак относятся статья Дэниела Дж. Бернштейна 2005 года «Understanding Brute Force» и статья Майкла Винера 2004 года «The Full Cost of Cryptanalytic Attacks»; обе бесплатно доступны в интернете.

Чтобы определить уровень стойкости для ключа заданного размера, посетите https://www.keylength.com. На этом сайте собраны рекомендации нескольких государственных учреждений относительно размеров ключей, а также порядок величины стойкости, гарантируемой в зависимости от размера открытых ключей.

Наконец, в качестве упражнения выберите приложение, например приложение для защищённого обмена сообщениями, и определите его криптографические схемы, длины ключей и соответствующие уровни стойкости. Нередко обнаруживаются удивительные несоответствия: например, первая схема обеспечивает 256-битный уровень стойкости, а вторая - лишь 100-битный. Стойкость всей системы зачастую не превосходит стойкости её самого слабого компонента.

Глава 4. Блочные шифры

Во время холодной войны Соединённые Штаты и Советский Союз разрабатывали собственные шифры. Правительство США создало Data Encryption Standard (DES), принятый в качестве федерального стандарта с 1979 по 2005 год, а КГБ разработал ГОСТ 28147-89 - алгоритм, остававшийся секретным до 1990 года и применяемый до сих пор. В 2000 году американский Национальный институт стандартов и технологий (NIST) выбрал преемника DES - Advanced Encryption Standard (AES), алгоритм, разработанный в Бельгии и теперь присутствующий в большинстве электронных устройств. AES, DES и ГОСТ 28147-89 являются блочными шифрами - разновидностью шифров, сочетающей базовый алгоритм, работающий с блоками данных, и режим работы, то есть метод обработки последовательностей блоков данных.

В этой главе рассматриваются базовые алгоритмы, лежащие в основе блочных шифров, обсуждаются режимы их работы и объясняется, как всё это действует совместно. Кроме того, мы разберём работу AES, а в заключение рассмотрим классическое средство атаки из 1970-х годов - атаку «встреча посередине» - и один из излюбленных методов атак 2000-х годов: оракулы заполнения.

Что такое блочный шифр?

Блочный шифр состоит из алгоритма шифрования и алгоритма расшифрования:

• Алгоритм шифрования (`E`) принимает ключ `K` и блок открытого текста `P`, а создаёт блок шифротекста `C`. Операция шифрования записывается как `C = E(K, P)`.

• Алгоритм расшифрования (`D`) является обратным алгоритму шифрования и расшифровывает сообщение до исходного открытого текста `P`. Эта операция записывается как `P = D(K, C)`.

Поскольку алгоритмы шифрования и расшифрования взаимно обратны, обычно они включают сходные операции.

Цели безопасности

Если вы следили за предыдущими рассуждениями о шифровании, случайности и неразличимости, определение безопасного блочного шифра вас не удивит. Мы продолжим определять безопасность, так сказать, как способность выглядеть случайным.

Чтобы блочный шифр был безопасен, он должен представлять собой псевдослучайную перестановку (pseudorandom permutation, PRP): пока ключ остаётся секретным, злоумышленник не должен уметь вычислить выход блочного шифра для любого входа. Иными словами, пока `K` секретен и с точки зрения злоумышленника случаен, злоумышленник не должен иметь ни малейшего представления о том, как выглядит `E(K, P)` для любого заданного `P`.

В более общем случае злоумышленники не должны уметь обнаруживать какие-либо закономерности во входных и выходных значениях блочного шифра. Иными словами, имея доступ по принципу чёрного ящика к функциям шифрования и расшифрования с некоторым фиксированным неизвестным ключом, должно быть невозможно отличить блочный шифр от истинно случайной перестановки. По той же причине злоумышленники не должны уметь восстанавливать секретный ключ безопасного блочного шифра; иначе с его помощью они отличили бы блочный шифр от случайной перестановки. Отсюда следует, что злоумышленники не могут

предсказать открытый текст, соответствующий заданному шифротексту, созданному блочным шифром.

Размер блока

Блочный шифр характеризуется двумя величинами: размером блока и размером ключа. Безопасность зависит от обеих. У большинства блочных шифров блоки имеют длину 64 или 128 бит: блоки DES содержат 64 (2^6) бита, а блоки AES - 128 (2^7) бит. В вычислительной технике размеры обычно измеряют степенями двойки, что упрощает обработку, хранение и адресацию данных. Но почему 2^6 и 2^7, а не 2^4 или 2^16 бит?

Важно, чтобы блоки блочного шифра не были слишком большими: это уменьшает и длину шифротекста, и объём занимаемой памяти. Сначала блочные шифры преобразуют входные данные в последовательность блоков. Поэтому, чтобы зашифровать 16-битное сообщение при 128-битных блоках, сообщение нужно превратить в 128-битный блок, который блочный шифр обработает и вернёт в виде 128-битного шифротекста. Чем шире блоки, тем больше эти накладные расходы. Для обработки 128-битного блока требуется не менее 128 бит памяти. Блоки длиной 64, 128 и даже 512 бит достаточно коротки, чтобы помещаться в регистрах большинства процессоров или реализовываться специализированными аппаратными схемами, благодаря чему в большинстве случаев возможны эффективные реализации. Но более крупные блоки, например длиной несколько килобайт, способны заметно повлиять на стоимость и производительность реализации.

Когда длина шифротекста или объём памяти критичны, возможно, придётся применять 64-битные блоки, поскольку они дают более короткие шифротексты и потребляют меньше памяти. В остальных случаях лучше использовать блоки длиной 128 бит или больше, главным образом потому, что современные процессоры зачастую обрабатывают 128-битные блоки эффективнее 64-битных, а сами они безопаснее (см. атаку Sweet32). В частности, процессоры могут задействовать инструкции для эффективной параллельной обработки одного или нескольких 128-битных блоков, например семейство инструкций Advanced Vector Extensions (AVX) в процессорах Intel.

Атака по кодовой книге

Блоки не должны быть слишком большими, но не должны быть и слишком маленькими; иначе они могут оказаться уязвимыми для атак по кодовой книге - атак на блочные шифры, эффективных лишь при малом размере блоков. Для 16-битных блоков атака по кодовой книге работает так:

1. Получить 65 536 (2^16) шифротекстов, соответствующих каждому 16-битному блоку открытого текста.

2. Построить таблицу поиска - кодовую книгу, - сопоставляющую каждому блоку шифротекста соответствующий блок открытого текста.

3. Чтобы расшифровать неизвестный блок шифротекста, найти в таблице соответствующий блок открытого текста.

При 16-битных блоках таблице поиска требуется всего 2^16 * 16 = 2^20 бит памяти, или 128 килобайт. При 32-битных блоках потребность в памяти возрастает до 16 гигабайт, что всё ещё приемлемо. Но при 64-битных блоках пришлось бы хранить 2^70 бит (один зеттабит, или 128 эксабайт), так что об этом можно забыть. Поэтому для более крупных блоков длиной 128 бит и больше атаки по кодовой книге проблемой не являются.

Как строят блочные шифры

Существуют сотни блочных шифров, но лишь несколько методов их построения. На практике блочный шифр - не один гигантский алгоритм, а повторение раундов: короткой последовательности операций, слабой самой по себе, но сильной при многократном применении. Существует два основных метода построения раунда: подстановочно-перестановочные сети, как в AES, и схемы Фейстеля, как в DES. В этом разделе вы познакомитесь с этими методами после рассмотрения атаки, которая работает, когда все раунды совпадают друг с другом.

Раунды блочного шифра

Вычисление блочного шифра сводится к вычислению последовательности раундов. Раунд в блочном шифре - базовое преобразование, простое в описании и реализации, которое повторяется несколько раз, образуя алгоритм блочного шифра. Конструкция из небольшого многократно повторяемого компонента проще для реализации и анализа, чем конструкция из одного огромного алгоритма.

Например, блочный шифр с тремя раундами шифрует открытый текст, вычисляя `C = R_3(R_2(R_1(P)))`, где `R_1`, `R_2` и `R_3` - раунды, а `P` - открытый текст. У каждого раунда также должно существовать обратное преобразование, чтобы получатель мог восстановить открытый текст. В частности, `P = iR_1(iR_2(iR_3(C)))`, где `iR_1` - преобразование, обратное `R_1`, и так далее.

Раундовые функции `R_1`, `R_2` и последующие обычно представляют собой один и тот же алгоритм, но параметризуются значением, называемым раундовым ключом. Две раундовые функции с разными раундовыми ключами будут вести себя по-разному и потому при одинаковых входных данных дадут разные результаты.

Раундовые ключи выводятся из главного ключа `K` с помощью алгоритма развёртывания ключа. Например, `R_1` принимает раундовый ключ `K_1`, `R_2` - раундовый ключ `K_2` и так далее.

Раундовые ключи должны различаться на каждом раунде. Более того, не все раундовые ключи должны совпадать с ключом `K`; иначе все раунды окажутся одинаковыми, и блочный шифр будет менее безопасен, как описано далее.

Атака скольжением и раундовые ключи

В блочном шифре ни один раунд не должен совпадать с другим, чтобы избежать атаки скольжением. Как показано на рис. 4-1, атака скольжением ищет две пары «открытый текст - шифротекст» `(P_1, C_1)` и `(P_2, C_2)`, где `P_2 = R(P_1)`, если `R` - раунд шифра. Когда раунды одинаковы, отношение между двумя открытыми текстами `P_2 = R(P_1)` влечёт отношение `C_2 = R(C_1)` между соответствующими шифротекстами. На рис. 4-1 показаны три раунда, но отношение `C_2 = R(C_1)` сохраняется при любом числе раундов: 3, 10 или 100. Проблема состоит в том, что знание входа и выхода одного раунда часто помогает восстановить ключ. (Подробности приведены в статье Алекса Бирюкова и Дэвида Вагнера 1999 года «Advanced Slide Attacks».)

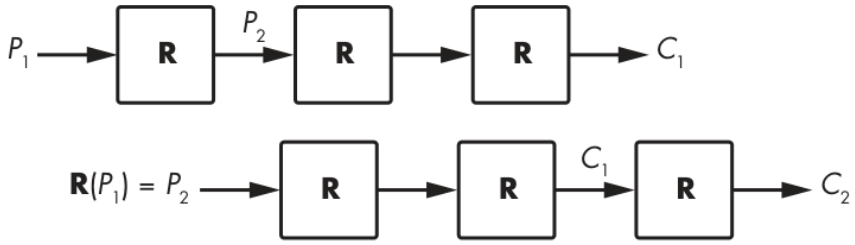

Рисунок 4-1. Принцип атаки скольжением на блочные шифры с одинаковыми раундами

Использование разных раундовых ключей в качестве параметров гарантирует, что раунды будут вести себя по-разному, и тем самым разрушает атаки скольжением.

> **Примечание.** Возможным побочным результатом и преимуществом применения раундовых ключей является защита от атак по побочным каналам, то есть атак, использующих сведения, просочившиеся из реализации шифра, например электромагнитное излучение. Если преобразование главного ключа `K` в раундовый ключ `K_i` необратимо, злоумышленник, обнаружив `K_i`, не сможет с его помощью найти `K`. К сожалению, лишь у немногих блочных шифров развёртывание ключа является односторонним. Например, развёртывание ключа AES позволяет злоумышленникам вычислить `K` по любому раундовому ключу `K_i`.

Подстановочно-перестановочные сети

Если вы читали учебники по криптографии, то, вероятно, сталкивались с понятиями смешивания и рассеивания. Смешивание означает, что входные данные, то есть открытый текст и ключ шифрования, подвергаются сложным преобразованиям, а рассеивание - что эти преобразования в равной степени зависят от всех входных битов. На высоком уровне смешивание относится к глубине, а рассеивание - к широте. При проектировании блочного шифра смешивание и рассеивание принимают форму операций подстановки и перестановки, объединяемых в подстановочно-перестановочные сети (substitution-permutation networks, SPN).

Подстановка часто реализуется в виде S-блоков, или блоков подстановки, - небольших таблиц поиска, преобразующих фрагменты по 4 или 8 бит. Например, первый из восьми S-блоков блочного шифра Serpent состоит из 16 элементов (`3 8 f 1 a 6 5 b e d 4 2 7 0 9 c`), каждый из которых представляет 4-битную тетраду. Этот S-блок отображает 4-битную тетраду `0000` в 3 (`0011`), 4-битную тетраду `0101` (5 в десятичной системе) - в 6 (`0110`) и так далее.

> **Примечание.** S-блоки необходимо тщательно выбирать, чтобы они были криптографически стойкими: они должны быть максимально нелинейными, то есть входы и выходы должны связываться сложными уравнениями, и не иметь статистического смещения, что, например, означает потенциальное влияние инвертирования любого входного бита на любой выходной бит.

Перестановка в подстановочно-перестановочной сети может сводиться к простому изменению порядка битов, которое легко реализовать, но которое не слишком сильно перемешивает биты. Вместо переупорядочивания битов в некоторых шифрах для их перемешивания применяют базовую линейную алгебру и умножение матриц: выполняется последовательность операций умножения на фиксированные значения (элементы матрицы), после чего результаты складываются. Такие операции способны быстро создать зависимости между всеми битами шифра

и тем самым обеспечить сильное рассеивание. Например, блочный шифр FOX преобразует четырёхбайтовый вектор `(a, b, c, d)` в `(a', b', c', d')`, определённый следующим образом:

```
a' = a + b + c + (2 * d),
b' = a + (253 * b) + (2 * c) + d,
c' = (253 * a) + (2 * b) + c + d,
d' = (2 * a) + b + (253 * c) + d.
```

В этих уравнениях числа 2 и 253 интерпретируются как двоичные многочлены, а не как целые числа, поэтому сложение и умножение определяются несколько иначе, чем мы привыкли. Например, вместо `2 + 2 = 4` получается `2 + 2 = 0`. Как бы то ни было, каждый байт начального состояния влияет на все четыре байта конечного состояния.

Схемы Фейстеля

В 1970-х годах инженер IBM Хорст Фейстель разработал блочный шифр Lucifer, работающий следующим образом:

1. Разделить 64-битный блок на две 32-битные половины `L` и `R`.

2. Присвоить `L` значение `L xor F(R)`, где `F` - подстановочно-перестановочный раунд.

3. Поменять значения `L` и `R` местами.

4. Перейти к шагу 2 и повторить его 15 раз.

5. Объединить `L` и `R` в выходной 64-битный блок.

Эта конструкция представляет собой схему Фейстеля, показанную на рис. 4-2. Слева изображена только что описанная схема; справа - функционально эквивалентное представление, где вместо перестановки `L` и `R` раунды поочерёдно выполняют операции `L = L xor F(R)` и `R = R xor F(L)`.

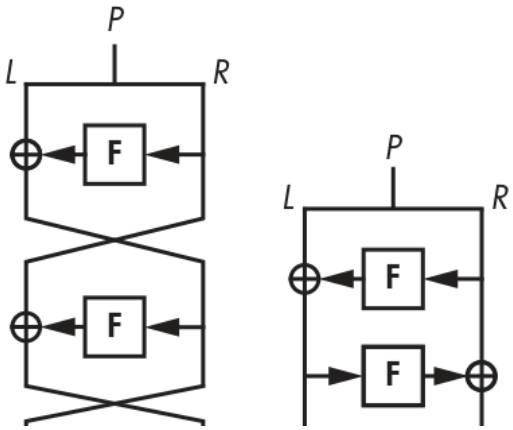

Рисунок 4-2. Построение блочного шифра по схеме Фейстеля в двух эквивалентных формах

Чтобы упростить схемы, я не показал на рис. 4-2 ключи, но обратите внимание: первая функция `F` принимает раундовый ключ `K_1`, а вторая - другой раундовый ключ `K_2`. В DES функции `F` принимают 48-битный раундовый ключ, выведенный из 56-битного ключа `K`.

В схеме Фейстеля функция `F` может быть либо псевдослучайной перестановкой (PRP), либо псевдослучайной функцией (pseudorandom function, PRF). PRP даёт разные выходы для любых двух разных входов, тогда как у PRF существуют значения `X` и `Y`, для которых `F(X) = F(Y)`. Но в схеме Фейстеля эта разница несущественна, пока `F` криптографически стойка.

Сколько раундов должно быть в схеме Фейстеля? DES выполняет 16 раундов, а ГОСТ 28147-89 - 32 раунда. Если функция `F` максимально стойка, в теории достаточно четырёх раундов, однако

реальные шифры используют больше раундов для защиты от возможных слабостей F.

Advanced Encryption Standard

AES - самый широко используемый шифр в мире. До принятия AES стандартным шифром был DES с его смехотворной 56-битной стойкостью, а также усовершенствованная версия DES, известная как Triple DES, или 3DES. Хотя 3DES обеспечивает более высокий уровень стойкости - 112 бит, - он неэффективен: для получения 112-битной стойкости требуется ключ длиной 168 бит, а программная реализация медленна (DES создавался для быстрой работы в интегральных схемах, а не на массовых процессорах). AES устраняет обе проблемы.

В 2000 году NIST стандартизировал AES в качестве замены DES, после чего AES фактически стал мировым стандартом шифрования. Сегодня AES поддерживает большинство коммерческих продуктов шифрования, а NSA одобрило его для защиты совершенно секретных сведений. (Некоторые страны предпочитают собственные шифры главным образом потому, что не хотят применять стандарт США, но в действительности AES скорее бельгийский, чем американский.)

> **Примечание.** Когда алгоритм участвовал в конкурсе AES как один из 15 кандидатов, он назывался Rijndael - это имя образовано из фамилий его создателей, Реймена и Дамена, и произносится примерно как «рейн-дал». Конкурс AES проводился NIST с 1997 по 2000 год, чтобы определить «несекретный, открыто опубликованный алгоритм шифрования, способный защищать конфиденциальные государственные сведения далеко в следующем столетии», как было сказано в объявлении конкурса 1997 года в Federal Register. Конкурс AES был своего рода состязанием талантов для криптографов, где любой желающий мог участвовать, представив шифр или взломав шифр другого участника.

Внутреннее устройство AES

AES обрабатывает блоки по 128 бит с помощью секретного ключа длиной 128, 192 или 256 бит. Чаще всего используется 128-битный ключ, поскольку с ним шифрование немного быстрее, а различие между 128- и 256-битной стойкостью не имеет значения для большинства применений.

Некоторые шифры работают с отдельными битами или 64-битными словами, а AES манипулирует байтами. Он рассматривает 16-байтовый открытый текст как двумерный массив байтов (`s = s_0, s_1, ..., s_15`), как показано на рис. 4-3. (Мы используем букву `s`, поскольку этот массив является внутренним состоянием, или просто состоянием.) AES преобразует байты, столбцы и строки этого массива, создавая конечное значение - шифротекст.

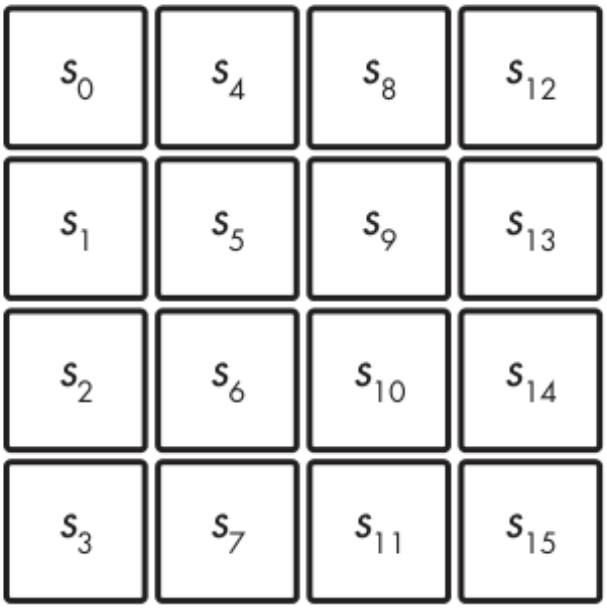

Рисунок 4-3. Внутреннее состояние AES в виде массива 4 x 4 из 16 байт

Для преобразования состояния AES использует структуру SPN, показанную на рис. 4-4: 10 раундов при 128-битном ключе, 12 - при 192-битном и 14 - при 256-битном.

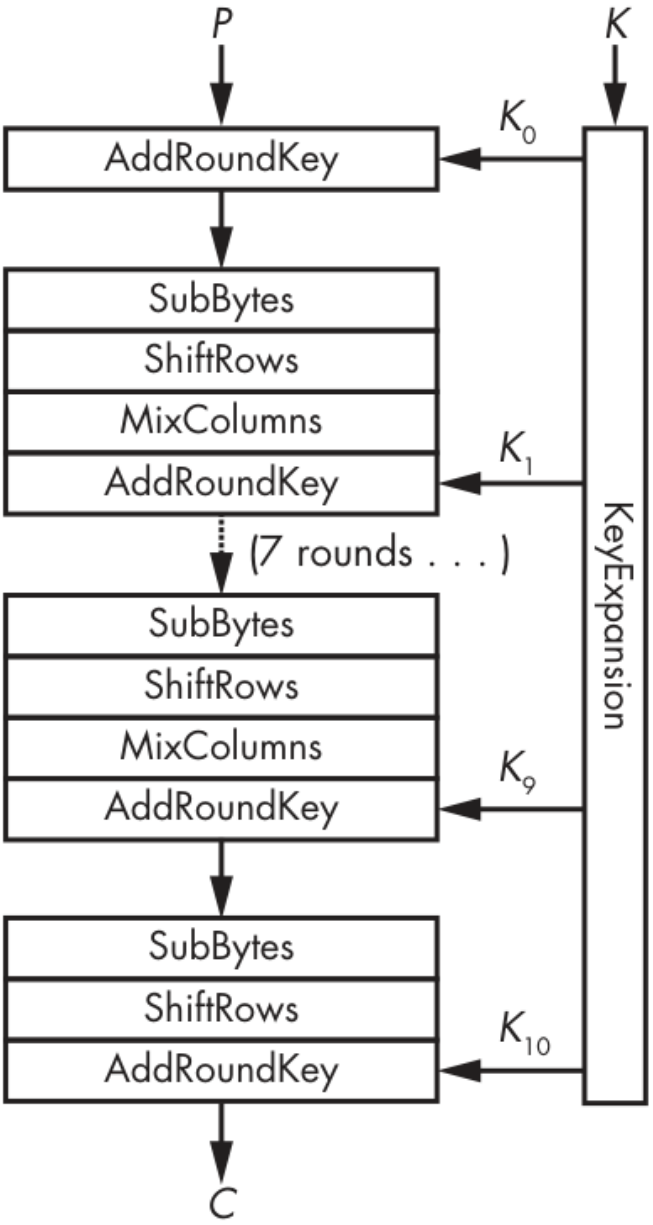

Рисунок 4-4. Внутренние операции AES

На рис. 4-4 показаны четыре строительных блока раунда AES. Обратите внимание: все раунды, кроме последнего, представляют собой последовательность SubBytes, ShiftRows, MixColumns и AddRoundKey:

• **AddRoundKey.** Выполняет XOR раундового ключа с внутренним состоянием.

• **SubBytes.** Заменяет каждый байт (`s_0, s_1, ..., s_15`) другим байтом согласно S-блоку. В данном случае S-блок представляет собой таблицу поиска из 256 элементов.

• **ShiftRows.** 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 четырёх столбцов состояния, то есть к каждой группе ячеек одинакового оттенка серого на левой стороне рис. 4-5.

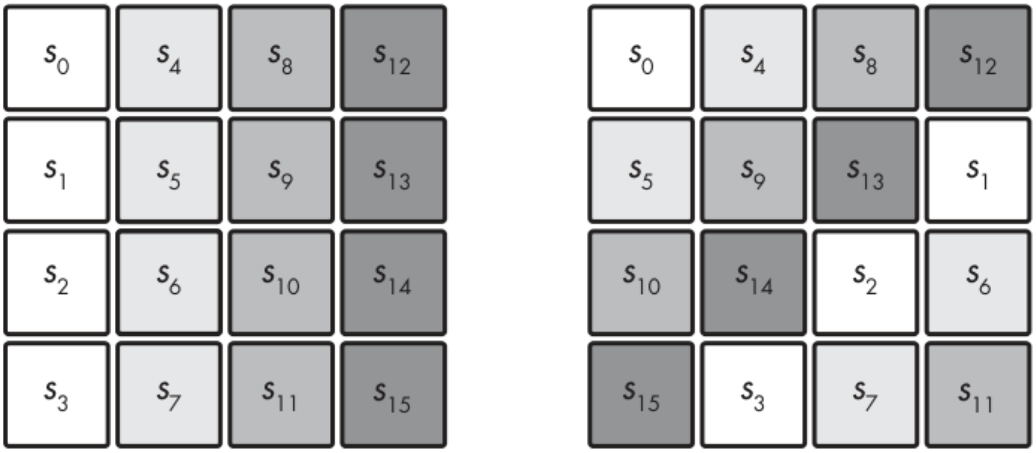

Рисунок 4-5. ShiftRows циклически сдвигает байты в каждой строке внутреннего состояния.

Помните, что в SPN буква S означает подстановку, а P - перестановку. Здесь слоем подстановки является SubBytes, а слоем перестановки - сочетание ShiftRows и MixColumns.

Функция развёртывания ключа KeyExpansion, показанная на рис. 4-4, является алгоритмом развёртывания ключа AES. Она создаёт из 16-байтового ключа 11 раундовых ключей (`K_0, K_1, ..., K_10`) по 16 байт каждый, используя тот же S-блок, что и SubBytes, и сочетание операций XOR. Важное свойство KeyExpansion состоит в том, что, имея любой раундовый ключ `K_i`, злоумышленник может определить все остальные раундовые ключи, а также главный ключ `K`, обратив алгоритм. Возможность получить ключ из любого раундового ключа снижает устойчивость шифра к атакам по побочным каналам, при которых злоумышленник может без труда восстановить раундовый ключ.

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

• Без KeyExpansion все раунды использовали бы один и тот же ключ `K`, и AES был бы уязвим для атак скольжением.

• Без AddRoundKey шифрование не зависело бы от ключа, поэтому любой мог бы расшифровать любой шифротекст без ключа.

• SubBytes добавляет нелинейные операции, повышающие криптографическую стойкость. Без неё AES представлял бы собой лишь большую систему линейных уравнений, которую можно решить методами школьной алгебры, а именно методом исключения Гаусса.

• Без ShiftRows изменения в одном столбце никогда не влияли бы на остальные столбцы, а значит, AES можно было бы взломать, построив для каждого столбца по четыре кодовые книги из `2^32` элементов. (Помните, что в безопасном блочном шифре инвертирование одного входного бита должно влиять на все выходные биты.)

• Без MixColumns изменения одного байта не влияли бы ни на какие другие байты состояния. Тогда злоумышленник с выбранным открытым текстом мог бы расшифровать любой шифротекст, сохранив 16 таблиц поиска по 256 байт каждая, содержащих зашифрованные значения каждого возможного значения байта.

Обратите внимание: на рис. 4-4 последний раунд AES не содержит операции MixColumns. Она опущена, чтобы избежать бесполезных вычислений: поскольку MixColumns линейна, эффект этой операции в самом последнем раунде можно устранить, комбинируя биты способом, не зависящим от их значений или ключа. Но обратить SubBytes невозможно, не зная значения состояния до

AddRoundKey.

Чтобы расшифровать шифротекст, AES разворачивает каждую операцию, применяя обратную функцию: обратная таблица поиска SubBytes отменяет преобразование SubBytes, ShiftRows выполняет сдвиг в противоположном направлении, применяется обратная MixColumns, то есть матрица, обратная матрице, задающей эту операцию, а XOR в AddRoundKey остаётся неизменным, поскольку операцией, обратной XOR, является другая XOR.

AES в действии

В качестве упражнения можно воспользоваться библиотекой `cryptography` в Python, чтобы зашифровать и расшифровать блок данных с помощью AES, как показано в листинге 4-1.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Выбираем случайный 16-байтовый ключ с помощью криптографического PRNG Python.
k = urandom(KEY_SIZE)
print(f"k = {k.hex()}")

# Создаём экземпляр AES-128.
aes = Cipher(algorithms.AES(k), modes.ECB())
aes_ecb_encryptor = aes.encryptor()

# Присваиваем открытому тексту p строку из одних нулей.
p = bytes([0x00] * BLOCK_SIZE)

# Шифруем открытый текст p в шифротекст c.
c = aes_ecb_encryptor.update(p) + aes_ecb_encryptor.finalize()
print(f"enc({p.hex()}) = {c.hex()}")

# Расшифровываем шифротекст c в открытый текст p.
aes_ecb_decryptor = aes.decryptor()
p = aes_ecb_decryptor.update(c) + aes_ecb_decryptor.finalize()
print(f"dec({c.hex()}) = {p.hex()}")
```

Листинг 4-1. Шифрование и расшифрование блока с помощью AES в Python

Выполнение этого сценария даёт примерно следующий результат:

```
$ ./aes_block.py
k = 2c6202f9a582668aa96d511862d8a279
enc(00000000000000000000000000000000) = 12b620bb5eddcde9a07523e59292a6d7
dec(12b620bb5eddcde9a07523e59292a6d7) = 00000000000000000000000000000000
```

У вас результаты будут другими, потому что при каждом новом запуске ключ выбирается случайно.

Как реализовать AES

Реальное программное обеспечение AES работает не так, как алгоритм на рис. 4-4. В коде AES промышленного уровня вы не найдёте последовательных вызовов функций `SubBytes()`, `ShiftRows()` и `MixColumns()`, поскольку это было бы неэффективно. Вместо этого быстрые программные реализации AES используют таблицы и собственные инструкции процессора.

Табличные реализации

В табличных реализациях AES последовательность SubBytes-ShiftRows-MixColumns заменяется сочетанием операций XOR и поисков в таблицах, жёстко заданных в программе и загружаемых в память во время выполнения. Это возможно потому, что MixColumns эквивалентна XOR четырёх 32-битных значений, каждое из которых зависит от одного байта состояния и от SubBytes. Поэтому можно построить четыре таблицы по 256 записей, по одной для каждого значения байта, и реализовать последовательность SubBytes-MixColumns поиском четырёх 32-битных значений с последующим выполнением их XOR.

Например, табличная реализация на C из набора OpenSSL выглядит как листинг 4-2.

```
/* 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];
--snip--
```

Листинг 4-2. Фрагмент табличной реализации AES на C в OpenSSL

Базовой табличной реализации шифрования AES требуются четыре таблицы общим объёмом 4 Кбайт, потому что каждая хранит 256 32-битных значений, занимающих `256 * 32 = 8192` бита, или 1 Кбайт. Для расшифрования нужны ещё четыре таблицы, то есть ещё 4 Кбайт. Однако существуют приёмы, позволяющие сократить объём с 4 Кбайт до 1 Кбайт и даже меньше.

Увы, табличные реализации уязвимы для атак по времени доступа к кешу, использующих различия времени, когда программа читает или записывает элементы кеш-памяти. Время доступа меняется в зависимости от относительного положения запрашиваемых элементов в кеш-памяти. Поэтому временные характеристики раскрывают сведения о получаемом элементе, а тот, в свою очередь, раскрывает сведения о задействованных секретных данных.

Атак по времени доступа к кешу трудно избежать. Очевидным решением было бы полностью отказаться от таблиц поиска и написать программу, время выполнения которой не зависит от входных данных, но сделать это, сохранив прежнюю скорость, почти невозможно. Поэтому производители микросхем выбрали радикальное решение: вместо потенциально уязвимого программного обеспечения они полагаются на аппаратуру.

Собственные инструкции

Собственные инструкции AES (AES native instructions, AES-NI) решают проблему атак по времени доступа к кешу на программные реализации AES. Чтобы понять принцип AES-NI, вспомните, как программное обеспечение выполняется аппаратурой: для запуска программы микропроцессор преобразует двоичный код в последовательность инструкций, выполняемых компонентами интегральной схемы. Например, ассемблерная инструкция `MUL` над двумя 32-битными значениями активирует транзисторы, реализующие 32-битный умножитель микропроцессора. При реализации криптографического алгоритма обычно задаётся сочетание базовых операций - сложений, умножений, XOR и так далее, - а микропроцессор активирует схемы сложения, умножения и XOR в предписанном порядке.

Собственные инструкции AES поднимают этот подход на совершенно иной уровень, предоставляя разработчикам специализированные ассемблерные инструкции, вычисляющие AES. Вместо кодирования раунда AES последовательностью ассемблерных инструкций при использовании AES-NI достаточно вызвать инструкцию `AESENC`, и микросхема вычислит раунд за вас. Собственные инструкции позволяют приказать процессору выполнить раунд AES, а не программировать раунды как сочетание базовых операций.

Типичная ассемблерная реализация AES с собственными инструкциями выглядит как листинг 4-3.

```
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
```

Листинг 4-3. Реализация AES-128 с собственными инструкциями AES

Этот код шифрует 128-битный открытый текст, изначально находящийся в регистре `xmm0`, предполагая, что в регистрах с `xmm5` по `xmm15` хранятся предварительно вычисленные раундовые ключи; каждая инструкция записывает результат в `xmm0`. Начальная инструкция `PXOR` выполняет XOR первого раундового ключа перед вычислением первого раунда, а заключительная инструкция `AESENCLAST` выполняет последний раунд несколько иначе, чем остальные: MixColumns опускается.

> **Примечание.** На платформах с собственными инструкциями AES работает примерно в десять раз быстрее; на момент написания книги к таким платформам относились практически все микропроцессоры ноутбуков, настольных компьютеров и серверов, а также большинство мобильных телефонов и планшетов. Хотя Intel впервые предложила инструкции AES в 2008 году, они доступны и в процессорах AMD, а большинство архитектур помимо x86 также имеют эквивалентные инструкции, аппаратно реализующие AES. Например, набор инструкций Armv8 содержит инструкции `AESSE`, вычисляющую SubBytes и ShiftRows, и `AESMS`, вычисляющую MixColumns.
>
> В микроархитектуре Intel Ice Lake задержка инструкции `AESENC` составляет три такта при обратной пропускной способности в половину такта: вызов `AESENC` завершается за три такта, а в каждом такте можно сделать два новых вызова инструкции. Фактически внутренняя структура микроопераций, выполняющих `AESENC`, позволяет начать вычисление новой инструкции до завершения предыдущей. Более того, архитектура Ice Lake использует векторизованную версию инструкций AES, позволяющую запускать несколько инструкций одновременно. Подробности приведены в статье Нира Друкера, Шая Герона и Влада Краснова «Making AES Great Again».

Для последовательного шифрования серии блоков выполнение десяти раундов занимает `3 * 10 = 30` тактов, или `30/16 = 1,875` такта на байт. При частоте 2 ГГц (`2 * 10^9` тактов в секунду) теоретическая максимальная пропускная способность составляет около 1 Гбайт/с. Если блоки можно обрабатывать параллельно, дожидаться завершения одного вызова `AESENC` перед запуском другого не требуется. В таком случае за такт можно выполнять два вызова `AESENC` и получать два результата, что даёт значительно более высокую теоретическую пропускную способность - свыше 10 Гбайт/с при 2 ГГц - в зависимости от размера данных и режима работы.

Безопасность AES

AES настолько безопасен, насколько вообще может быть безопасен блочный шифр, и взломан он не будет никогда. В основе безопасности AES лежит тот факт, что все выходные биты некоторым сложным псевдослучайным образом зависят от всех входных битов. Чтобы добиться этого, создатели AES тщательно выбрали каждый компонент для определённой цели: MixColumns - за максимальные свойства рассеивания, SubBytes - за оптимальную нелинейность. Такая композиция защищает AES от целых классов криптоаналитических атак.

Но доказательства невосприимчивости AES ко всем возможным атакам не существует. Во-первых, мы не знаем всех возможных атак и не всегда умеем доказать безопасность шифра против конкретной атаки. Единственный способ обрести настоящую уверенность в безопасности AES - привлечь к атакам множество людей: пусть многие квалифицированные специалисты попытаются взломать AES и, в идеале, потерпят неудачу.

После более чем 15 лет и сотен исследовательских публикаций мы лишь слегка прикоснулись к теоретической безопасности AES. В 2011 году криптоаналитики нашли способ восстановить ключ AES-128, выполнив около `2^126` операций вместо `2^128`, то есть ускорив перебор в четыре раза. Но этой «атаке» требуется огромное число пар «открытый текст - шифротекст» - около `2^88` битов данных. Это интересный результат, но беспокоиться о нём не стоит.

При реализации и развёртывании криптографии следует заботиться о миллионе вещей, но безопасность AES к ним не относится. Наибольшая угроза блочным шифрам заключена не в базовых алгоритмах, а в режимах работы. Если выбран неверный режим или неправильно использован правильный, не спасёт даже такой сильный шифр, как AES.

Режимы работы

В главе 1 я объяснил, как схемы шифрования сочетают перестановку с режимом работы для обработки сообщений любой длины. В этом разделе рассматриваются основные режимы работы блочных шифров, их свойства безопасности и функциональности, а также правильные и неправильные способы применения. Начнём с самого глупого режима: электронной кодовой книги.

Режим электронной кодовой книги

Самый простой режим шифрования блочным шифром - электронная кодовая книга (electronic codebook, ECB), которую вообще едва ли можно назвать режимом работы. ECB принимает блоки открытого текста `P_1, P_2, ..., P_N` и обрабатывает каждый независимо, вычисляя `C_1 = E(K, P_1)`, `C_2 = E(K, P_2)` и так далее, как показано на рис. 4-6. Операция проста, но небезопасна: ECB небезопасен, и использовать его не следует.

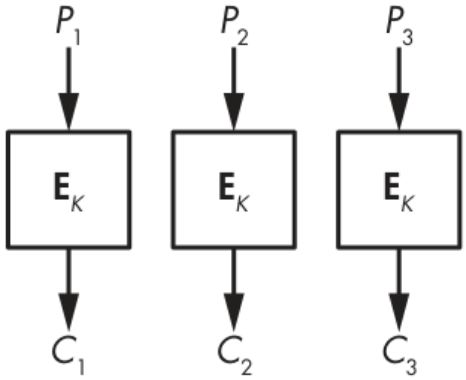

Рисунок 4-6. Режим ECB

Криптограф Microsoft Марш Рэй однажды сказал: «Все знают, что режим ECB плох, потому что мы видим пингвина». Он имел в виду знаменитую иллюстрацию небезопасности ECB с изображением талисмана Linux - пингвина Такса, как на рис. 4-7.

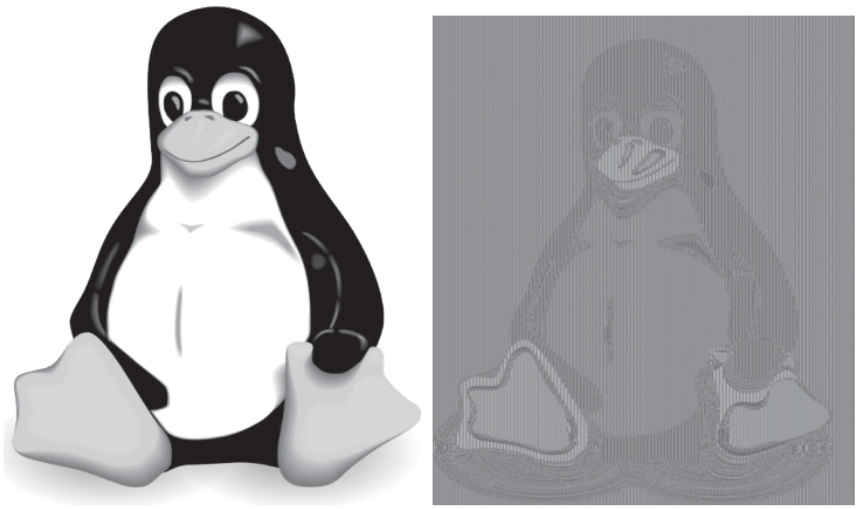

Рисунок 4-7. Исходное изображение (слева) и изображение, зашифрованное в ECB (справа)

Исходное изображение Такса находится слева, а зашифрованное с помощью AES в режиме ECB изображение - справа, хотя конкретный базовый шифр значения не имеет. В зашифрованной версии легко различить форму пингвина, потому что ECB зашифровал все блоки одного оттенка серого на исходном изображении в один и тот же новый оттенок серого на новом. Иными словами, шифрование ECB даёт то же изображение в других цветах.

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. Обратите внимание: шифрование даёт два одинаковых блока, а повторное шифрование с тем же ключом и тем же открытым текстом снова создаёт те же два блока.

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

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Функция blocks() разбивает строку данных на блоки, разделённые пробелами.
def blocks(data):
    split = [data[i:i+BLOCK_SIZE].hex() for i in range(0, len(data), BLOCK_SIZE)]
    return ' '.join(split)

k = urandom(KEY_SIZE)
print(f"k = {k.hex()}")

# Создаём экземпляр AES-128 для шифрования и расшифрования.
aes = Cipher(algorithms.AES(k), modes.ECB())
aes_ecb_encryptor = aes.encryptor()

# Присваиваем открытому тексту p два блока нулей.
p = bytes([0x00] * 2 * BLOCK_SIZE)

# Шифруем открытый текст p в шифротекст c.
c = aes_ecb_encryptor.update(p) + aes_ecb_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")
```

Листинг 4-4. Использование AES в режиме ECB в Python

Выполнение этого сценария даёт следующие блоки шифротекста:

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

При использовании режима ECB одинаковые блоки шифротекста раскрывают злоумышленнику одинаковые блоки открытого текста независимо от того, находятся ли эти блоки в одном шифротексте или в разных. Это показывает, что блочные шифры в режиме ECB не являются семантически безопасными.

Другая проблема ECB состоит в том, что режим принимает только полные блоки данных. Поэтому при 16-байтовых блоках, как у AES, можно шифровать лишь фрагменты длиной 16, 32, 48 байт или любое другое число байтов, кратное 16. Существует несколько способов решить эту проблему, как вы увидите на примере следующего режима CBC. (Я не стану рассказывать, как эти приёмы работают с ECB, потому что ECB вообще не следует использовать.)

Режим сцепления блоков шифротекста

Сцепление блоков шифротекста (cipher block chaining, CBC) похоже на ECB, но с небольшим изменением, которое приводит к большой разнице: вместо шифрования `i`-го блока `P_i` как `C_i = E(K, P_i)` режим CBC задаёт `C_i = E(K, P_i xor C_(i-1))`, где `C_(i-1)` - предыдущий блок шифротекста; тем самым блоки `C_(i-1)` и `C_i` сцепляются. При шифровании первого блока `P_1` предыдущего блока шифротекста ещё нет, поэтому CBC принимает случайное начальное значение (initial value, IV), как показано на рис. 4-8.

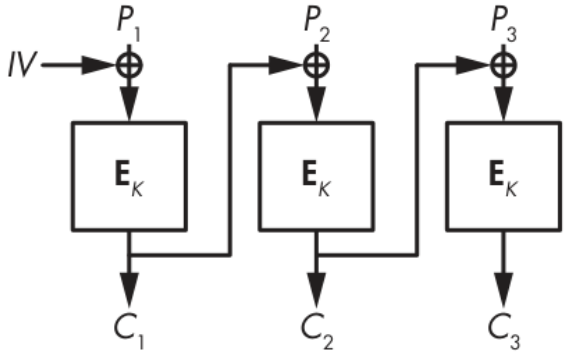

Рисунок 4-8. Режим CBC

Режим CBC делает каждый блок шифротекста зависимым от всех предыдущих блоков и гарантирует, что одинаковые блоки открытого текста не станут одинаковыми блоками шифротекста. Случайное начальное значение гарантирует, что два одинаковых открытых текста будут зашифрованы в разные шифротексты, если два вызова шифра используют два разных начальных значения.

Листинг 4-5 иллюстрирует оба преимущества. Программа принимает 32-байтовое сообщение из одних нулей, как в листинге 4-4, дважды шифрует его в CBC и показывает два шифротекста. Выделенная в оригинале строка `iv = urandom(BLOCK_SIZE)` выбирает новое случайное IV для каждого нового шифрования.

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

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Функция blocks() разбивает строку данных на блоки, разделённые пробелами.
def blocks(data):
    split = [data[i:i+BLOCK_SIZE].hex() for i in range(0, len(data), BLOCK_SIZE)]
    return ' '.join(split)

# Выбираем случайный ключ.
k = urandom(KEY_SIZE)
print(f"k  = {k.hex()}")

# Выбираем случайное IV.
iv = urandom(BLOCK_SIZE)
print(f"iv = {iv.hex()}")

# Выбираем экземпляр AES в режиме CBC.
aes_cbc_encryptor = Cipher(algorithms.AES(k), modes.CBC(iv)).encryptor()

# Присваиваем открытому тексту p два блока нулей.
p = bytes([0x00] * 2 * BLOCK_SIZE)

c = aes_cbc_encryptor.update(p) + aes_cbc_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")

# Теперь используем другое IV и тот же ключ.
iv = urandom(BLOCK_SIZE)
print(f"iv = {iv.hex()}")

aes_cbc_encryptor = Cipher(algorithms.AES(k), modes.CBC(iv)).encryptor()
c = aes_cbc_encryptor.update(p) + aes_cbc_encryptor.finalize()
print(f"enc({blocks(p)}) = {blocks(c)}")
```

Листинг 4-5. Использование AES в режиме CBC

Два открытых текста одинаковы и состоят из двух нулевых блоков, но зашифрованные блоки должны различаться, как в следующем примере выполнения:

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

Увы, CBC часто используют с постоянным, а не случайным IV, что раскрывает одинаковые открытые тексты и открытые тексты, начинающиеся с одинаковых блоков. Например, предположим, что CBC шифрует двухблочный открытый текст `P_1 || P_2` в двухблочный шифротекст `C_1 || C_2`. Если CBC с тем же IV шифрует `P_1 || P_2'`, где `P_2'` - блок, отличный от `P_2`, получившийся шифротекст будет выглядеть как `C_1 || C_2'`, где `C_2'` отличается от `C_2`, но первый блок `C_1` совпадает. Поэтому злоумышленник может догадаться, что первый блок обоих открытых текстов одинаков, хотя видит лишь шифротексты.

> **Примечание.** Для расшифрования в режиме CBC необходимо знать IV, использованное при шифровании, поэтому IV передаётся вместе с шифротекстом в открытом виде.

Благодаря параллелизму расшифрование CBC может быть значительно быстрее шифрования. При шифровании нового блока `P_i` необходимо дождаться предыдущего блока `C_(i-1)`, а расшифрование блока вычисляет `P_i = D(K, C_i) xor C_(i-1)`, и предыдущий блок открытого текста `P_(i-1)` не требуется. Это означает, что все блоки можно расшифровывать параллельно и одновременно, если известен предыдущий блок шифротекста, а обычно он известен.

Шифрование сообщений в режиме CBC

Вернёмся к проблеме завершения блока и рассмотрим обработку открытого текста, длина которого не кратна длине блока. Например, как зашифровать 18-байтовый открытый текст с помощью AES-CBC при 16-байтовых блоках? Что делать с оставшимися двумя байтами? Мы рассмотрим два широко применяемых решения. Первое, заполнение, делает шифротекст немного длиннее открытого текста, а второе, похищение шифротекста, создаёт шифротекст той же длины, что и открытый текст.

Заполнение сообщения

Заполнение - метод, позволяющий зашифровать сообщение любой длины, даже короче одного блока. Стандарт PKCS#7 и RFC 5652 определяют заполнение для блочных шифров, применяемое почти везде, где используется CBC.

С помощью заполнения сообщение расширяют до полного блока, добавляя к открытому тексту дополнительные байты. Для 16-байтовых блоков действуют следующие правила:

• Если остаётся 1 байт, например длина открытого текста равна 1, 17 или 33 байтам, сообщение дополняется 15 байтами `0f` (15 в десятичной системе).

• Если остаётся 2 байта, сообщение дополняется 14 байтами `0e` (14 в десятичной системе).

• Если остаётся 3 байта, сообщение дополняется 13 байтами `0d` (13 в десятичной системе).

Если имеется 15 байт открытого текста и для заполнения блока не хватает одного байта, добавляется единственный байт `01`. Если длина открытого текста уже кратна 16 - длине блока,

добавляется 16 байтов `10` (16 в десятичной системе). Этот приём обобщается на любую длину блока до 255 байт; для более крупных блоков одного байта недостаточно, чтобы закодировать значения больше 255.

Расшифрование заполненного сообщения работает так:

1. Расшифровать все блоки, как в CBC без заполнения.

2. Убедиться, что последние байты последнего блока соответствуют правилу заполнения: оканчиваются как минимум одним байтом `01`, как минимум двумя байтами `02`, как минимум тремя байтами `03` и так далее. Если заполнение недопустимо, например последние байты равны `01 02 03`, сообщение отклоняется. В противном случае расшифрование удаляет байты заполнения и возвращает оставшиеся байты открытого текста.

Один из недостатков заполнения состоит в том, что оно увеличивает шифротекст как минимум на один байт и максимум на целый блок.

Похищение шифротекста

Похищение шифротекста - ещё один приём шифрования сообщений, длина которых не кратна размеру блока. Он сложнее и менее популярен, чем заполнение, но даёт несколько преимуществ:

• Открытые тексты могут иметь любую длину в битах, а не только целое число байтов. Например, можно зашифровать 131-битное сообщение.

• Длина шифротекста в точности совпадает с длиной открытого текста.

• Похищение шифротекста не уязвимо для атак на оракул заполнения - мощных атак, иногда действующих против CBC с заполнением, как вы увидите в разделе «Атаки на оракул заполнения».

В режиме CBC при похищении шифротекста последний неполный блок открытого текста расширяется битами предыдущего блока шифротекста, а получившийся блок затем шифруется. Последний неполный блок шифротекста составляется из первых битов предыдущего блока шифротекста, то есть тех битов, которые не были добавлены к последнему блоку открытого текста.

На рис. 4-9 показаны три блока, последний из которых, `P_3`, неполон и обозначен нулём. Если `P_3` имеет длину 3 байта, мы выполняем его XOR с последними 12 байтами предыдущего блока шифротекста `E(K, P_2)` и возвращаем зашифрованный результат как `C_2`. Последний блок шифротекста `C_3` затем состоит из первых четырёх байтов `E(K, P_2)`. Расшифрование представляет собой простое обращение этой операции.

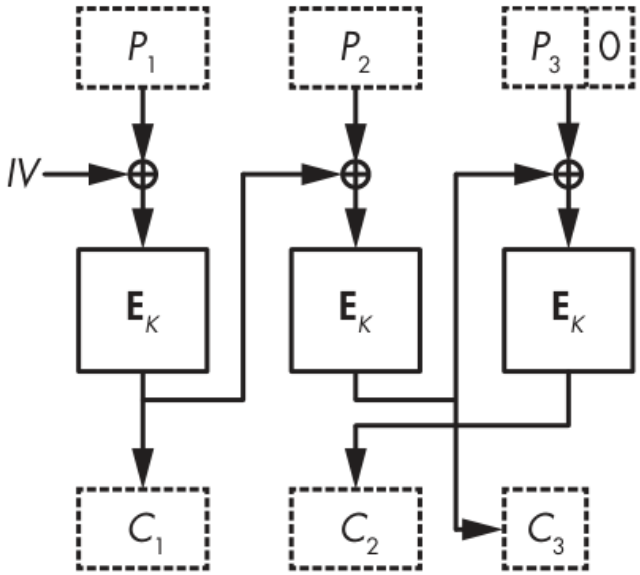

Рисунок 4-9. Похищение шифротекста при шифровании в режиме CBC

У похищения шифротекста нет серьёзных проблем, но этот метод неизящен, и реализовать его правильно трудно, особенно учитывая, что стандарт NIST определяет три разных способа реализации (см. Special Publication 800-38A).

Режим счётчика

Чтобы избежать проблем и сохранить преимущества похищения шифротекста, используйте режим счётчика (counter mode, CTR). CTR едва ли можно назвать режимом блочного шифра: он превращает блочный шифр в потоковый, который просто принимает биты и выдаёт биты, не обременяя себя понятием блоков. (Потоковые шифры подробно рассматриваются в главе 5.)

В режиме CTR (см. рис. 4-10) алгоритм блочного шифра не преобразует данные открытого текста. Вместо этого он шифрует блоки, составленные из счётчика и nonce. Счётчик - целое число, увеличиваемое для каждого блока. В пределах одного сообщения никакие два блока не должны использовать одинаковый счётчик, но разные сообщения могут применять одну и ту же последовательность значений счётчика (1, 2, 3, ...). Nonce - число, используемое только один раз. Оно одинаково для всех блоков одного сообщения, но никакие два сообщения не должны использовать одно и то же nonce.

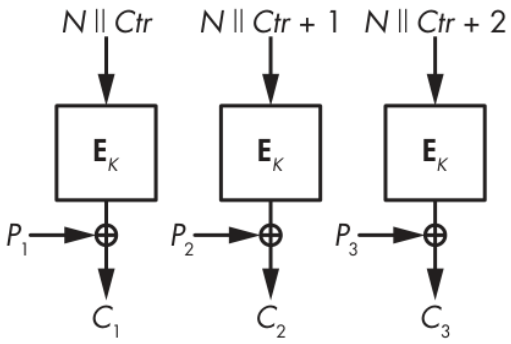

Рисунок 4-10. Режим CTR

На рис. 4-10 показано, что при шифровании в режиме CTR выполняется XOR открытого текста с потоком, полученным «шифрованием» nonce `N` и счётчика `Ctr`. Расшифрование выполняется так же, поэтому и для шифрования, и для расшифрования нужен только алгоритм шифрования. Практический пример приведён в сценарии Python из листинга 4-6.

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

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from os import urandom

BLOCK_SIZE = 16
KEY_SIZE = 16

# Выбираем случайный ключ.
k = urandom(KEY_SIZE)
print(f"k  = {k.hex()}")

# И случайное nonce
# (соблюдайте осторожность со случайными nonce, см. обсуждение ниже).
n = urandom(BLOCK_SIZE)
print(f"nonce  = {n.hex()}")

# Создаём 7-байтовый открытый текст p.
p = bytes([0x00] * 7)

# Шифруем открытый текст p с помощью AES-CTR.
aes_ctr_encryptor = Cipher(algorithms.AES(k), modes.CTR(n)).encryptor()

c = aes_ctr_encryptor.update(p) + aes_ctr_encryptor.finalize()
print(f"enc({p.hex()}) = {c.hex()}")

# Расшифровываем шифротекст c.
aes_ctr_decryptor = Cipher(algorithms.AES(k), modes.CTR(n)).decryptor()
p = aes_ctr_decryptor.update(c) + aes_ctr_decryptor.finalize()
print(f"dec({c.hex()}) = {p.hex()}")

# Расшифровываем шифротекст c с помощью функции шифрования.
aes_ctr_encryptor = Cipher(algorithms.AES(k), modes.CTR(n)).encryptor()
p = aes_ctr_encryptor.update(c) + aes_ctr_encryptor.finalize()
print(f"enc({c.hex()}) = {p.hex()}")
```

Листинг 4-6. Использование AES в режиме CTR

В этом примере выполнения четырёхбайтовый открытый текст шифруется в четырёхбайтовый шифротекст. Затем шифротекст расшифровывается с помощью функции шифрования:

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

Как и начальное значение CBC, nonce режима CTR задаётся шифрующей стороной и отправляется вместе с шифротекстом в открытом виде. Но в отличие от начального значения CBC nonce CTR не обязано быть случайным; оно должно быть лишь уникальным. Nonce должно быть уникальным по той же причине, по которой нельзя повторно использовать одноразовый блокнот: если при вызове псевдослучайного потока `S` с одним nonce зашифровать `P_1` в `C_1 = P_1 xor S`, а `P_2` - в `C_2 = P_2 xor S`, то `C_1 xor C_2` раскроет `P_1 xor P_2`.

Случайное nonce решает задачу, только если оно достаточно длинное. Например, если длина nonce равна `n` битам, после `2^(n/2)` шифрований и такого же числа nonce, скорее всего, появятся повторы. Для случайного nonce 64 бит недостаточно, поскольку повторения можно ожидать приблизительно после `2^32` nonce, а это неприемлемо малое число.

Уникальность счётчика гарантируется, если он увеличивается для каждого нового открытого текста и достаточно длинен, например имеет длину 64 бита.

Особое преимущество CTR состоит в возможности работать быстрее любого другого режима. Он не только допускает распараллеливание: шифрование можно начать ещё до получения сообщения, выбрав nonce и вычислив поток, с которым позднее будет выполнен XOR открытого текста.

> **Примечание.** В зависимости от реализуемой версии CTR nonce, передаваемое API как аргумент, можно конкатенировать со счётчиком, как на рис. 4-10, либо непосредственно использовать счётчик шириной в целый блок.

Что может пойти не так

Необходимо знать две атаки на блочные шифры: атаку «встреча посередине» - метод, открытый в 1970-х годах, но по-прежнему применяемый во многих криптоаналитических атаках и не имеющий отношения к атаке «человек посередине», - и атаку на оракул заполнения, класс атак, открытый академическими криптографами в 2002 году, затем по большей части проигнорированный и наконец вновь открытый десятилетие спустя вместе с несколькими уязвимыми приложениями.

Атаки «встреча посередине»

Блочный шифр 3DES - усовершенствованная версия стандарта DES 1970-х годов, принимающая ключ длиной `56 * 3 = 168` бит, что лучше 56-битного ключа DES. Но из-за атаки «встреча посередине» (meet-in-the-middle, MitM) уровень стойкости 3DES равен 112, а не 168 битам.

На рис. 4-11 показано, как 3DES шифрует блок с помощью функций шифрования и расшифрования DES: сначала выполняется шифрование с ключом `K_1`, затем расшифрование с ключом `K_2` и, наконец, шифрование с другим ключом `K_3`. Если `K_1 = K_2`, первые два вызова взаимно уничтожаются, и 3DES сводится к одному DES с ключом `K_3`. В 3DES выполняется последовательность «шифрование - расшифрование - шифрование», а не тройное шифрование, чтобы при необходимости системы могли имитировать DES через новый интерфейс 3DES.

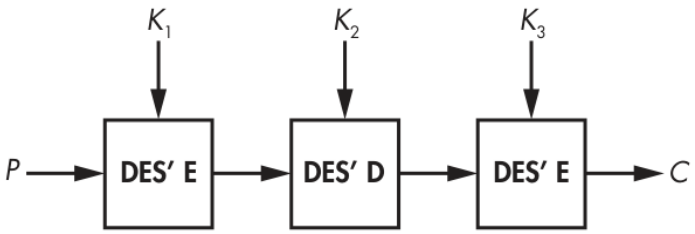

Рисунок 4-11. Конструкция блочного шифра 3DES

Почему используется тройной DES, а не просто двойной DES, то есть почему открытый текст `P` не шифруется в `E(K_2, E(K_1, P))`? Оказывается, из-за атаки MitM двойной DES лишь настолько же безопасен, как одинарный. Атака MitM показана на рис. 4-12.

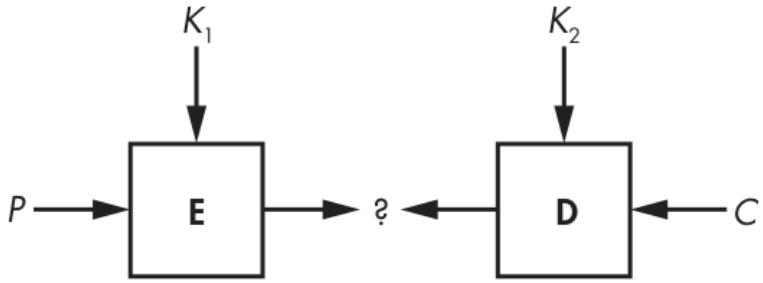

Рисунок 4-12. Атака MitM

Атака на двойной DES методом MitM работает следующим образом:

1. Предположим, у вас имеются `P` и `C = E(K_2, E(K_1, P))` с двумя неизвестными 56-битными ключами `K_1` и `K_2`. (DES принимает 56-битные ключи, поэтому в двойном DES суммарно 112 бит ключа.) Вы строите таблицу «ключ - значение» из `2^56` записей `E(K_1, P)`, где в качестве значения хранится `K_1`.

2. Для всех `2^56` значений `K_2` вычисляете `D(K_2, C)` и проверяете, присутствует ли полученное значение в таблице как индекс, то есть как среднее значение, обозначенное вопросительным знаком на рис. 4-12.

3. Если среднее значение обнаружено как индекс таблицы, извлекаете из таблицы соответствующий `K_1` и по другим парам `P` и `C` проверяете, что найденная пара `(K_1, K_2)` верна. Зашифруйте `P` с помощью `K_1` и `K_2`, а затем проверьте, совпадает ли полученный шифротекст с заданным `C`.

Этот метод восстанавливает `K_1` и `K_2`, выполняя около `2^57` вместо `2^112` операций: на шаге 1 шифруются `2^56` блоков, затем на шаге 2 расшифровывается не более `2^56` блоков, что в сумме даёт `2^56 + 2^56 = 2^57` операций. Кроме того, необходимо хранить `2^56` элементов по 15 байт, или около одного эксабайта. Это много, но существует приём, позволяющий выполнить ту же атаку с пренебрежимо малым объёмом памяти, как вы увидите в главе 6.

Атаку MitM можно применить к 3DES почти так же, как к двойному DES, за исключением того, что на третьем этапе перебираются все `2^112` сочетаний `K_2` и `K_3`. Вся атака завершается примерно за `2^112` операций, поэтому при 168 битах ключевого материала 3DES обеспечивает лишь 112-битную стойкость.

Атаки на оракул заполнения

Эту главу завершает одна из простейших и вместе с тем самых разрушительных атак 2000-х годов - атака на оракул заполнения. Помните, что заполнение дополняет открытый текст лишними байтами до целого блока. Например, открытый текст длиной 111 байт представляет собой последовательность из шести 16-байтовых блоков, за которыми следуют 15 байт. В таком случае для образования полного блока заполнение добавляет байт `01`. К 110-байтовому открытому тексту добавляются два байта `02`, к 109-байтовому - три байта `03` и так далее, вплоть до случая, когда добавляются 16 байтов `10`, где шестнадцатеричное значение `10` равно 16.

Оракул заполнения - система, поведение которой различается в зависимости от допустимости заполнения шифротекста, зашифрованного в CBC. Его можно представить как чёрный ящик или API, возвращающий либо успех, либо ошибку. Например, оракулом заполнения может служить служба на удалённом узле, отправляющая сообщения об ошибках при получении неправильно сформированных шифротекстов. Имея такой оракул, атаки на оракул заполнения фиксируют, какие входные данные имеют допустимое заполнение, а какие нет, и затем используют эти сведения для расшифрования выбранных значений шифротекста.

Предположим, вы хотите расшифровать блок шифротекста `C_2`. Обозначим через `X` искомое значение `D(K, C_2)`, а через `P_2` - блок, полученный после расшифрования в режиме CBC (см. рис. 4-13). Если выбрать случайный блок `C_1` и отправить оракулу двухблочный шифротекст `C_1 || C_2`, расшифрование завершится успешно только в том случае, если `C_1 xor X = P_2` оканчивается допустимым заполнением: одним байтом `01`, двумя байтами `02`, тремя байтами `03` и так далее.

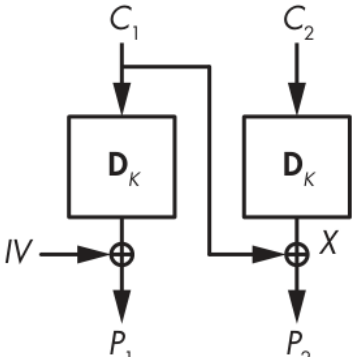

Рисунок 4-13. Атака на оракул заполнения восстанавливает X, выбирая C1 и проверяя допустимость заполнения.

Исходя из этого наблюдения, атака на оракул заполнения в режиме CBC может расшифровать блок `C_2` следующим образом. Байты обозначаются как элементы массива: `C1[0]` - первый байт `C_1`, `C1[1]` - второй и так далее до `C1[15]`, последнего байта `C_1`.

1. Выберите случайный блок `C_1` и изменяйте его последний байт, пока оракул заполнения не примет шифротекст как допустимый. Обычно в допустимом шифротексте `C_1[15] xor X[15] = 01`, поэтому `X[15]` будет найден приблизительно после перебора 128 значений `C_1[15]`.

2. Найдите значение `X[14]`, присвоив `C1[15]` значение `X[15] xor 02` и подбирая `C1[14]`, дающий правильное заполнение. Когда оракул принимает шифротекст как допустимый, это означает, что найдено значение `C1[14]`, при котором `C_1[14] xor X[14] = 02`.

3. Повторите шаги 1 и 2 для всех 16 байтов.

Атаке в среднем требуется 128 запросов к оракулу для каждого из 16 байтов, то есть всего около 2000 запросов. (Обратите внимание: каждый запрос должен использовать одно и то же начальное значение.)

Примечание. На практике реализовать атаку на оракул заполнения несколько сложнее, чем описано здесь, поскольку на шаге 1 приходится учитывать ошибочные догадки. Шифротекст может иметь допустимое заполнение не потому, что `P_2` оканчивается единственным байтом `01`, а потому, что оканчивается двумя байтами `02` или тремя байтами `03`. С этим можно справиться, проверяя допустимость шифротекстов, в которых изменено больше байтов.

Дополнительная литература

О блочных шифрах можно рассказать очень многое: и о работе алгоритмов, и о способах атак на них. Например, сети Фейстеля и SPN - не единственные способы построения блочного шифра. В блочных шифрах IDEA и FOX используется конструкция Лая-Мэсси, а в Threefish - сети ARX, сочетающие сложение, циклические сдвиги слов и XOR.

Помимо ECB, CBC и CTR существует множество других режимов. Некоторые режимы представляют собой традиционные приёмы, которыми никто не пользуется, например CFB и OFB, а другие предназначены для конкретных применений: XTS - для настраиваемого шифрования, GCM - для аутентифицированного шифрования.

Я рассказал о Rijndael, победителе конкурса AES, но в состязании участвовало ещё 14 алгоритмов: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, Magenta, MARS, RC6, SAFER+, Serpent и Twofish. Рекомендую найти сведения о них и узнать, как они работают, как были спроектированы, каким атакам подвергались и насколько быстро работают. Стоит также изучить разработки NSA - Skipjack и более современные SIMON и SPECK - и новые «облегчённые» блочные шифры, такие как GIFT, KATAN, PRESENT или PRINCE.

Глава 5. Потоковые шифры

Симметричные шифры делятся на блочные и потоковые. Вспомним из главы 4, что блочные шифры смешивают фрагменты битов открытого текста с битами ключа, создавая фрагменты шифротекста того же размера, обычно 64 или 128 бит. Потоковые шифры, напротив, не смешивают биты открытого текста и ключа; вместо этого они создают из ключа псевдослучайные биты и шифруют открытый текст, выполняя его XOR с псевдослучайными битами, подобно одноразовому блокноту, описанному в главе 1.

Потоковых шифров иногда избегают, поскольку исторически они были хрупче блочных и чаще взламывались: и экспериментальные шифры, созданные любителями, и шифры, развёрнутые в системах, которыми пользуются миллионы людей, включая мобильные телефоны, Wi-Fi и смарт-карты общественного транспорта. Но, к счастью, хотя на это ушло почти 20 лет, теперь мы умеем проектировать безопасные потоковые шифры и доверяем им защиту соединений Bluetooth, мобильной связи 4G и соединений TLS.

Сначала в этой главе рассматривается работа потоковых шифров и обсуждаются два основных класса: шифры с состоянием и шифры на основе счётчика. Затем мы изучим потоковые шифры, ориентированные на аппаратуру и программное обеспечение, рассмотрим небезопасные шифры A5/1, применявшийся в мобильной связи GSM, и RC4 из старых версий TLS, а также современные безопасные шифры Grain-128a для аппаратуры и Salsa20 для программного обеспечения.

Как работают потоковые шифры

Потоковые шифры больше похожи на детерминированные генераторы случайных битов (DRBG), чем на блочные шифры, поскольку создают поток псевдослучайных битов, а не непосредственно смешивают данные открытого текста.

От DRBG потоковые шифры отличаются тем, что DRBG принимает одно входное значение, а потоковый шифр - два: ключ и nonce. Ключ должен оставаться секретным и обычно имеет длину 128 или 256 бит. Nonce не обязано быть секретным, но должно быть уникальным для каждого ключа; обычно его длина составляет от 64 до 128 бит.

Потоковые шифры создают псевдослучайный поток битов, называемый ключевым потоком. Для шифрования ключевой поток складывается по XOR с открытым текстом, а для расшифрования снова по XOR с шифротекстом. На рис. 5-1 показана базовая операция шифрования потоковым шифром, где SC - алгоритм потокового шифра, KS - ключевой поток, `P` - открытый текст, а `C` - шифротекст.

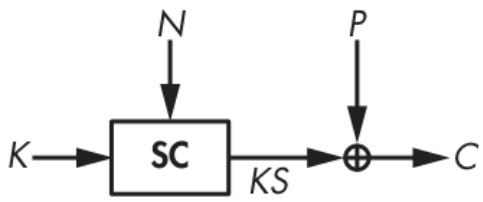

Рисунок 5-1. Потоковый шифр принимает секретный ключ K и открытое nonce N и выполняет шифрование.

Потоковый шифр вычисляет `KS = SC(K, N)`, шифрует как `C = P xor KS` и расшифровывает как `P = C xor KS`. Функции шифрования и расшифрования совпадают, потому что выполняют одно и то же:

XOR битов с ключевым потоком. Поэтому, например, некоторые криптографические библиотеки предоставляют одну функцию `encrypt` и для шифрования, и для расшифрования.

Потоковые шифры позволяют зашифровать сообщение с ключом `K_1` и nonce `N_1`, а затем другое сообщение с тем же ключом `K_1` и nonce `N_2`, отличным от `N_1`, либо с ключом `K_2`, отличным от `K_1`, и nonce `N_1`. Но повторно шифровать с `K_1` и `N_1` нельзя, поскольку один и тот же ключевой поток `KS` будет использован дважды. Иными словами, получится первый шифротекст `C_1 = P_1 xor KS` и второй `C_2 = P_2 xor KS`, а если известно `P_1`, можно определить `P_2 = C_1 xor C_2 xor P_1`.

> **Примечание.** Nonce - сокращение от «number used only once», то есть «число, используемое только один раз». В контексте потоковых шифров его иногда называют IV, или «начальным значением».

На высоком уровне потоковые шифры делятся на два типа: с состоянием и на основе счётчика. У потокового шифра с состоянием имеется секретное внутреннее состояние, которое развивается по мере создания ключевого потока. Шифр инициализирует состояние из ключа и nonce, а затем вызывает функцию обновления, чтобы изменить значение состояния и получить из него один или несколько битов ключевого потока, как показано на рис. 5-2. Например, RC4 является шифром с состоянием, а Salsa20 основан на счётчике.

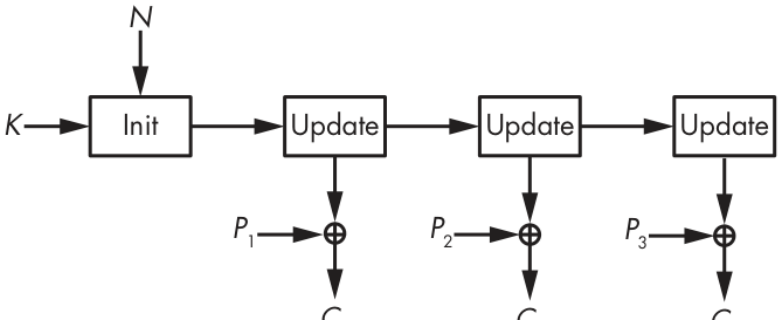

Рисунок 5-2. Потоковый шифр с состоянием

Потоковые шифры на основе счётчика создают фрагменты ключевого потока из ключа, nonce и значения счётчика, как на рис. 5-3. В отличие от потоковых шифров с состоянием такие шифры, как Salsa20, не отслеживают секретное состояние во время создания ключевого потока, если не считать значение счётчика.

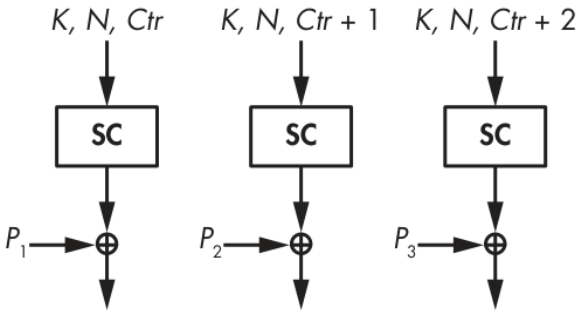

Рисунок 5-3. Потоковый шифр на основе счётчика

Эти два подхода определяют высокоуровневую архитектуру потокового шифра независимо от работы базовых алгоритмов. Внутреннее устройство потокового шифра также относится к одной из двух категорий в зависимости от целевой платформы: ориентированное на аппаратуру или на программное обеспечение.

Потоковые шифры, ориентированные на аппаратуру

Говоря об аппаратуре, криптографы имеют в виду специализированные интегральные схемы (application-specific integrated circuits, ASIC), программируемые логические устройства (programmable logic devices, PLD) и программируемые пользователем вентильные матрицы (field-programmable gate arrays, FPGA). Аппаратная реализация шифра представляет собой электронную схему, которая реализует криптографический алгоритм на уровне битов и не может использоваться ни для чего другого; иными словами, это специализированная аппаратура. Программные реализации криптографических алгоритмов, напротив, лишь сообщают микропроцессору, какие инструкции выполнить для запуска алгоритма. Эти инструкции оперируют байтами или словами, а затем вызывают части электронных схем, реализующие операции общего назначения, такие как сложение и умножение. Программное обеспечение работает с байтами или словами длиной 32 либо 64 бита, тогда как аппаратура работает с битами. Первые потоковые шифры работали с битами, чтобы обойтись без сложных операций над словами и тем самым быть эффективнее в аппаратуре, которая в то время была их целевой платформой.

Потоковые шифры главным образом использовались в аппаратных реализациях, поскольку обходились дешевле блочных. Им требовалось меньше памяти и логических вентилей, чем блочным шифрам, поэтому они занимали меньшую площадь интегральной схемы и снижали стоимость производства. Например, если измерять площадь стандартным для интегральных схем числом эквивалентных вентилей, можно было найти потоковые шифры, занимавшие менее 1000 эквивалентных вентилей; напротив, типичным блочным шифрам, ориентированным на программное обеспечение, требовалось как минимум 10 000 эквивалентных вентилей, что делало криптографию на порядок дороже, чем с потоковыми шифрами.

Однако сегодня блочные шифры уже не дороже потоковых: во-первых, появились блочные шифры, удобные для аппаратной реализации и примерно такие же малые, как потоковые; во-вторых, стоимость аппаратуры резко упала. Тем не менее потоковые шифры часто связывают с аппаратурой, поскольку когда-то они были лучшим вариантом.

В следующем разделе я объясню базовый механизм аппаратных потоковых шифров, называемый регистрами сдвига с обратной связью (feedback shift registers, FSR). Почти все аппаратные потоковые шифры так или иначе опираются на FSR: и шифр A5/1, использовавшийся в мобильных телефонах 2G, и более современный Grain-128a.

> **Примечание.** Первый стандартный блочный шифр Data Encryption Standard (DES) был оптимизирован для аппаратуры, а не для программного обеспечения. Когда в 1970-х годах правительство США стандартизировало DES, большинство целевых применений были аппаратными. Поэтому неудивительно, что S-блоки DES малы и быстро вычисляются в аппаратной логической схеме, но неэффективны в программном обеспечении. В отличие от DES современный Advanced Encryption Standard (AES) работает с байтами и потому эффективнее DES в программных реализациях.

Регистры сдвига с обратной связью

Бесчисленное множество потоковых шифров использует FSR, поскольку они просты и хорошо изучены. FSR - массив битов с функцией обновления обратной связи, которую я буду обозначать `f`. Состояние FSR хранится в массиве, или регистре, а каждое обновление FSR использует функцию обратной связи для изменения значения состояния и выдачи одного выходного бита.

На практике FSR работает так: если `R_0` - начальное значение FSR, следующее состояние `R_1` определяется как `R_0`, сдвинутое влево на один бит; покидающий регистр бит возвращается как

выходной, а освободившаяся позиция заполняется значением `f(R_0)`.

То же правило повторяется для вычисления последующих состояний `R_2`, `R_3` и так далее. Иными словами, если `R_t` - состояние FSR в момент `t`, следующее состояние `R_(t+1)` равно:

```
R_(t+1) = (R_t << 1) | f(R_t).
```

В этом уравнении `|` - оператор логического OR, а `<<` - оператор сдвига, как в языке C. Например, для 8-битной строки `00001111` получаем:

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

Сдвиг перемещает биты влево, отбрасывая крайний левый бит для сохранения длины состояния и обнуляя крайний правый. Операция обновления FSR идентична, только крайний правый бит устанавливается не в 0, а в `f(R_t)`.

Рассмотрим, например, четырёхбитный FSR, функция обратной связи `f` которого выполняет XOR всех четырёх битов. Инициализируем состояние следующим значением:

```
1 1 0 0
```

Теперь сдвинем биты влево; единица выдаётся на выход, а крайний правый бит получает следующее значение:

```
f(1100) = 1 xor 1 xor 0 xor 0 = 0.
```

Теперь состояние выглядит так:

```
1 0 0 0
```

Следующее обновление выдаёт 1, сдвигает состояние влево и устанавливает крайний правый бит в следующее значение:

```
f(1000) = 1 xor 0 xor 0 xor 0 = 1.
```

Теперь состояние выглядит так:

```
0 0 0 1
```

Следующие три обновления возвращают три нулевых бита и дают состояния:

```
0 0 1 1
0 1 1 0
1 1 0 0
```

Таким образом, после пяти итераций мы возвращаемся к начальному состоянию `1100`; пять обновлений состояния из любого значения, наблюдавшегося в этом цикле, снова возвращают нас к этому начальному значению. Мы говорим, что 5 - это период FSR для любого из значений `1100`, `1000`, `0001`, `0011` и `0110`. Поскольку период равен 5, десять тактов дают одну и ту же пятибитную последовательность дважды. Если тактировать FSR двадцать раз, начиная с состояния `1100`, он выдаст следующую последовательность:

```
11000110001100011000
```

Это всего лишь последовательность `11000`, повторенная четыре раза. Следует избегать FSR, выдающих повторяющиеся шаблоны: как правило, чем больше период, тем лучше.

> **Примечание.** Если вы собираетесь использовать FSR в потоковом шифре, избегайте регистров с короткими периодами, которые делают выход более предсказуемым. Для некоторых типов FSR период легко определить, но для других сделать это почти невозможно.

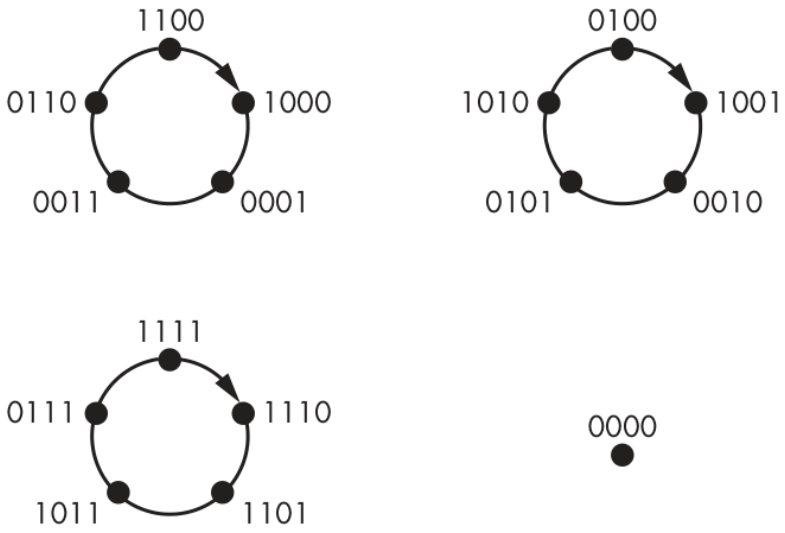

Рисунок 5-4. Циклы состояний FSR, функция обратной связи которого вычисляет XOR четырех битов

Действительно, у этого FSR имеются ещё два цикла периода 5: `{0100, 1001, 0010, 0101, 1010}` и `{1111, 1110, 1101, 1011, 0111}`. Обратите внимание, что каждое состояние может принадлежать только одному циклу состояний. Здесь имеются три цикла по пять состояний, охватывающие 15 из `2^4 = 16` возможных значений нашего четырёхбитного регистра. Шестнадцатое возможное значение, `0000`, как показано на рис. 5-4, образует цикл периода 1, поскольку FSR преобразует `0000` в `0000`.

По существу, FSR - это регистр битов, каждое обновление которого выдаёт один бит, крайний левый бит регистра, а функция вычисляет новый крайний правый бит регистра. Все остальные биты сдвигаются влево. Период FSR из некоторого начального состояния - это число обновлений, необходимое для того, чтобы FSR снова вошёл в то же состояние. Если для этого требуется `N` обновлений, FSR будет вновь и вновь выдавать одни и те же `N` битов.

Линейные регистры сдвига с обратной связью

Линейные регистры сдвига с обратной связью (linear feedback shift registers, LFSR) - это FSR с линейной функцией обратной связи, а именно функцией, вычисляющей XOR некоторых битов состояния, как в примере четырёхбитного FSR из предыдущего раздела, функция обратной связи которого возвращала XOR всех четырёх битов регистра. Вспомним, что в криптографии линейность означает предсказуемость и указывает на простую лежащую в основе математическую структуру. Как и можно было ожидать, благодаря этой линейности LFSR можно анализировать с помощью таких понятий, как линейная сложность, конечные поля и примитивные многочлены, однако я опущу математические подробности и приведу лишь основные факты.

> **Примечание.** В линейной алгебре линейным преобразованием `f` называется функция, удовлетворяющая равенству `f(u + v) = f(u) + f(v)`. Если известны `f(u)` и `f(v)`, можно определить `f(u + v)`, не зная `u` или `v`. С нелинейной функцией всё значительно сложнее: по `f(u)` и `f(v)` нельзя легко найти `f(u + v)`.

Для периода LFSR решающее значение имеет выбор битов, которые складываются посредством XOR. Позиции можно выбрать так, чтобы гарантировать максимальный период `2^n-1` для `n`-битного LFSR. Чтобы описывать выбранные позиции, пронумеруем биты от 1 для крайнего правого до `n` для

крайнего левого и сопоставим LFSR многочлен

```
1 + X + X^2 + ... + X^n,
```

включая член `X^i` тогда и только тогда, когда `i`-й бит входит в XOR функции обратной связи. Период LFSR максимален тогда и только тогда, когда этот многочлен примитивен. Примитивный многочлен обладает, в частности, следующими свойствами:

• Он неприводим, то есть не раскладывается в произведение многочленов меньшей степени. Например, `X+X^3` не является неприводимым, поскольку

```
X+X^3=(1+X)(X+X^2).
```

• Он удовлетворяет и другим свойствам, которые трудно объяснить без дополнительного математического аппарата, но легко проверить алгоритмически.

> **Примечание.** Максимальный период `n`-битного LFSR равен `2^n - 1`, а не `2^n`, поскольку состояние из одних нулей всегда бесконечно зацикливается само на себе. Поскольку XOR любого числа нулей равен нулю, новые биты, поступающие в состояние из функции обратной связи, всегда будут нулевыми; следовательно, состояние из одних нулей обречено навсегда оставаться нулевым.

Например, четырехбитному LFSR на рисунке 5-5 соответствует многочлен `1+X+X^3+X^4`: для формирования нового значения `L_1` посредством XOR складываются биты в позициях 1, 3 и 4. Этот многочлен не примитивен, поскольку раскладывается как `(1+X^3)(1+X)`.

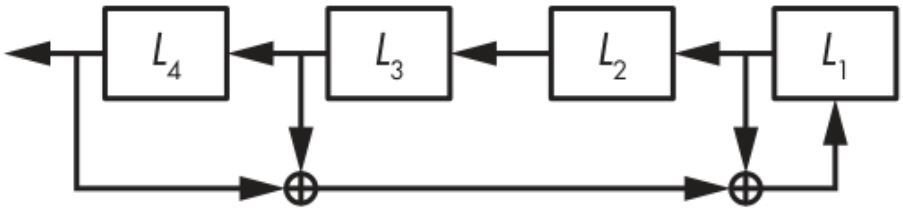

Рисунок 5-5. LFSR с многочленом `1+X+X^3+X^4`

Поэтому период данного LFSR не максимален. Если начать с состояния `0001`, после сдвига и вычисления обратной связи получится `0011`, а дальнейшие состояния будут следующими:

```
0111
1100
1000
0001
```

После шести обновлений LFSR возвращается к начальному состоянию, поэтому его период равен 6.

На рисунке 5-6 показан другой четырехбитный LFSR. Ему соответствует примитивный многочлен `1+X^3+X^4`, а потому он имеет максимальный период. Начиная с состояния `0001`, он проходит следующую последовательность состояний:

```
0001
0010
0100
1001
0011
0110
1101
1010
0101
1011
0111
1111
1110
1100
1000
0001
```

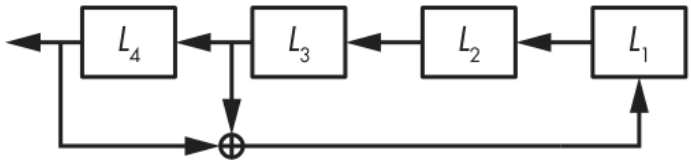

Рисунок 5-6. LFSR с примитивным многочленом `1+X^3+X^4`

Состояние проходит по всем возможным значениям, кроме `0000`, не повторяясь, пока в конце концов не зациклится. Это показывает, что период максимален, а многочлен обратной связи примитивен.

Увы, использовать LFSR как потоковый шифр небезопасно. Если `n` - длина LFSR в битах, злоумышленнику достаточно всего `n` выходных битов, чтобы восстановить начальное состояние LFSR, после чего он сможет определить все предыдущие биты и предсказать все будущие. Эта атака возможна потому, что LFSR линеен, а значит, отношения между битами состояния подчиняются линейным уравнениям, которые легко решать. С помощью алгоритма Берлекэмпа - Мэсси можно решить уравнения, заданные математической структурой LFSR, и найти не только его начальное состояние, но и многочлен обратной связи. Более того, для успеха даже не требуется знать точную длину LFSR: алгоритм Берлекэмпа - Мэсси можно повторять для всех возможных значений `n`, пока не встретится правильное.

Итак, LFSR криптографически слабы из-за своей линейности. Выходные биты и биты начального состояния связаны простыми короткими уравнениями, которые можно решить методами линейной алгебры школьного уровня. Чтобы усилить LFSR, добавим щепотку нелинейности.

Фильтрованные LFSR

Чтобы устранить небезопасность LFSR, его линейность скрывают, пропуская выходные биты через нелинейную функцию до их выдачи; так получается фильтрованный LFSR, показанный на рис. 5-7.

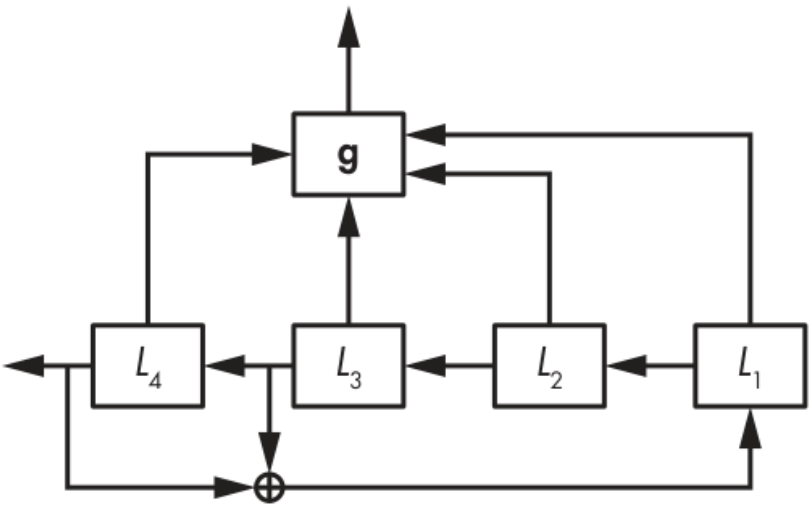

Рисунок 5-7. Фильтрованный LFSR

Функция g должна быть нелинейной. Помимо XOR, в ней обычно используются операции AND или OR. Например, функция

```
L_1 L_2 + L_3 L_4
```

означает `(L_1 & L_2) xor (L_3 & L_4)` в обозначениях языка C; знак умножения между переменными в алгебраической записи опускается.

> **Примечание.** Функции обратной связи можно записывать либо непосредственно через биты FSR, как `L_1 L_2 + L_3 L_4`, либо в эквивалентной многочленной записи `1 + X X^2 + X^3 X^4`. Непосредственную запись легче понять, но многочленная лучше подходит для математического анализа свойств FSR. Если математические свойства не важны, мы будем придерживаться непосредственной записи.

Фильтрованные LFSR сильнее обычных LFSR, поскольку их нелинейная функция препятствует прямолинейным атакам. Тем не менее систему взламывают более сложные атаки следующих типов:

• **Алгебраические атаки** составляют по наблюдаемым выходным битам систему нелинейных уравнений, неизвестными в которой служат биты внутреннего состояния, а затем решают ее.
• **Кубические атаки** вычисляют производные булевой функции, чтобы снизить ее степень до первой, после чего решают получившуюся линейную систему.
• **Быстрые корреляционные атаки** используют то, что поведение фильтрующей функции может быть близко к поведению некоторой линейной функции.

Урок прост: пластырем пулевое отверстие не заделать. Добавление заплат к сломанному алгоритму не делает его безопасным; исправлять нужно саму основу конструкции.

Нелинейные регистры сдвига с обратной связью

Нелинейные регистры сдвига с обратной связью (nonlinear feedback shift registers, NFSR) устроены так же, как LFSR, но используют нелинейную функцию обратной связи, включающую, например, операции AND или OR. У этого подхода есть как преимущества, так и недостатки.

Одно из преимуществ нелинейных функций обратной связи состоит в том, что они делают NFSR криптографически сильнее LFSR: выходные биты сложным образом зависят от начального секретного состояния согласно уравнениям экспоненциального размера. Линейная функция LFSR сохраняет отношения простыми, не более чем с n членами `(N_1, N_2, ..., N_n)`, если `N_i` - биты

состояния NFSR. Например, четырёхбитный NFSR с начальным секретным состоянием (N_1,N_2,N_3,N_4) и функцией обратной связи N_1 + N_2 + N_1N_2 + N_3N_4 выдаёт первый выходной бит, равный

```
N_1+N_2+N_1N_2+N_3N_4.
```

На второй итерации значение N_1 заменяется этим новым битом. Выражая второй выходной бит через начальное состояние, получаем уравнение

```
(N_1+N_2+N_1N_2+N_3N_4)+N_1
+(N_1+N_2+N_1N_2+N_3N_4)N_1+N_2N_3
=N_1+N_2+N_1N_2+N_2N_3+N_3N_4+N_1N_3N_4.
```

Новое уравнение имеет алгебраическую степень 3 - это наибольшее число перемноженных битов, здесь в N_1N_3N_4, - вместо степени 2 у функции обратной связи, а также шесть членов вместо четырёх. Поэтому повторное применение нелинейной функции быстро порождает неуправляемые уравнения: размер выхода растёт экспоненциально. При работе NFSR эти уравнения никогда не вычисляются, но злоумышленнику приходится решать их, чтобы взломать систему.

Один из недостатков NFSR состоит в том, что не существует эффективного способа определить период NFSR или узнать, максимален ли он. Для n-битного NFSR, чтобы проверить максимальность его периода, требуется выполнить почти 2^n испытаний. Для больших NFSR длиной 80 бит и более такой расчет невозможен.

К счастью, есть прием, позволяющий применять NFSR, не опасаясь коротких периодов: можно объединить LFSR и NFSR и получить как гарантированный максимальный период, так и криптографическую стойкость. Именно так работает Grain-128a.

Grain-128a

Помните конкурс AES, который обсуждался в главе 4 в связи с блочным шифром AES? Потоковый шифр Grain появился благодаря похожему проекту - конкурсу eSTREAM. Конкурс завершился в 2008 году публикацией короткого списка рекомендованных потоковых шифров, включавшего четыре аппаратно-ориентированных и четыре программно-ориентированных шифра. Grain относится к аппаратным шифрам, а Grain-128a представляет собой усовершенствованную версию, созданную авторами исходного Grain. Механизм работы Grain-128a показан на рисунке 5-8.

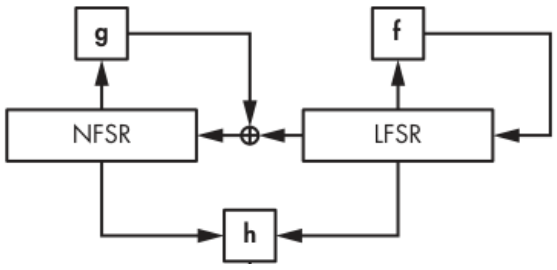

Рисунок 5-8. Механизм Grain-128a со 128-битным NFSR и 128-битным LFSR

Grain-128a устроен примерно настолько просто, насколько вообще может быть устроен потоковый шифр: он объединяет 128-битный LFSR, 128-битный NFSR и фильтрующую функцию h. LFSR имеет максимальный период 2^128-1, благодаря чему период всей системы составляет не менее 2^128-1 и система защищена от возможных коротких циклов в NFSR. В то же время NFSR и нелинейная фильтрующая функция h добавляют криптографическую стойкость.

Grain-128a принимает 128-битный ключ и 96-битный одноразовый номер. Он копирует 128 битов ключа в 128 битов NFSR, а 96 битов одноразового номера - в первые 96 битов LFSR, заполняя оставшиеся 32 разряда единицами и одним нулевым битом в самом конце. На этапе инициализации вся система обновляется 256 раз, прежде чем вернуть первый бит ключевого потока. Во время инициализации бит, возвращаемый функцией h, не выдается как часть ключевого потока, а поступает в LFSR, чтобы его последующее состояние зависело и от ключа, и от одноразового номера.

Функция обратной связи LFSR в Grain-128a имеет вид

```
f(L)=L_32+L_47+L_58+L_90+L_121+L_128,
```

где `L_1,L_2,...,L_128` - биты LFSR. Эта функция обратной связи использует всего 6 битов из 128-битного LFSR, но этого достаточно для получения примитивного многочлена, гарантирующего максимальный период. Малое число битов снижает стоимость аппаратной реализации.

Вот многочлен обратной связи NFSR в Grain-128a, где состояние обозначено (`N_1,...,N_128`):

```
g(N)=() N_32+N_37+N_72+N_102+N_128+N_44N_60+N_61N_125
+N_63N_67+N_69N_101+N_80N_88+N_110N_111+N_115N_117
+N_46N_50N_58+N_103N_104N_106+N_33N_35N_36N_40.
```

Эта функция была тщательно выбрана так, чтобы максимизировать криптографическую стойкость и одновременно минимизировать стоимость реализации. Ее алгебраическая степень равна 4, поскольку член с наибольшим числом переменных содержит четыре переменные, а именно `N_33N_35N_36N_40`. Кроме того, функцию g нельзя аппроксимировать линейной функцией, поскольку она обладает высокой нелинейностью. Наконец, помимо вычисления g, Grain-128a складывает посредством XOR выходной бит LFSR с результатом обратной связи и подает полученное значение в качестве нового крайнего правого бита NFSR.

Фильтрующая функция h - еще одна нелинейная функция. Она принимает 9 битов NFSR и 7 битов LFSR и объединяет их способом, обеспечивающим хорошие криптографические свойства.

На момент написания книги неизвестно ни одной атаки на Grain-128a, и я уверен, что он останется безопасным. Grain-128a применяется в некоторых недорогих встраиваемых системах, которым нужен компактный и быстрый потоковый шифр, - обычно в закрытых промышленных системах. Именно поэтому Grain-128a мало известен сообществу разработчиков программного обеспечения с открытым исходным кодом.

A5/1

A5/1 - потоковый шифр, применявшийся для шифрования голосовой связи в стандарте мобильной связи 2G. Стандарт A5/1 был создан в 1987 году, но опубликован лишь в конце 1990-х после того, как его восстановили методами обратной разработки. В начале 2000-х появились атаки, и в конце концов A5/1 был взломан способом, позволяющим выполнять реальную, а не только теоретическую расшифровку защищенной связи. Посмотрим, почему и как это произошло.

Механизм A5/1

Как показано на рисунке 5-9, A5/1 опирается на три LFSR и использует прием, который на первый взгляд кажется хитроумным, но на деле не обеспечивает безопасности.

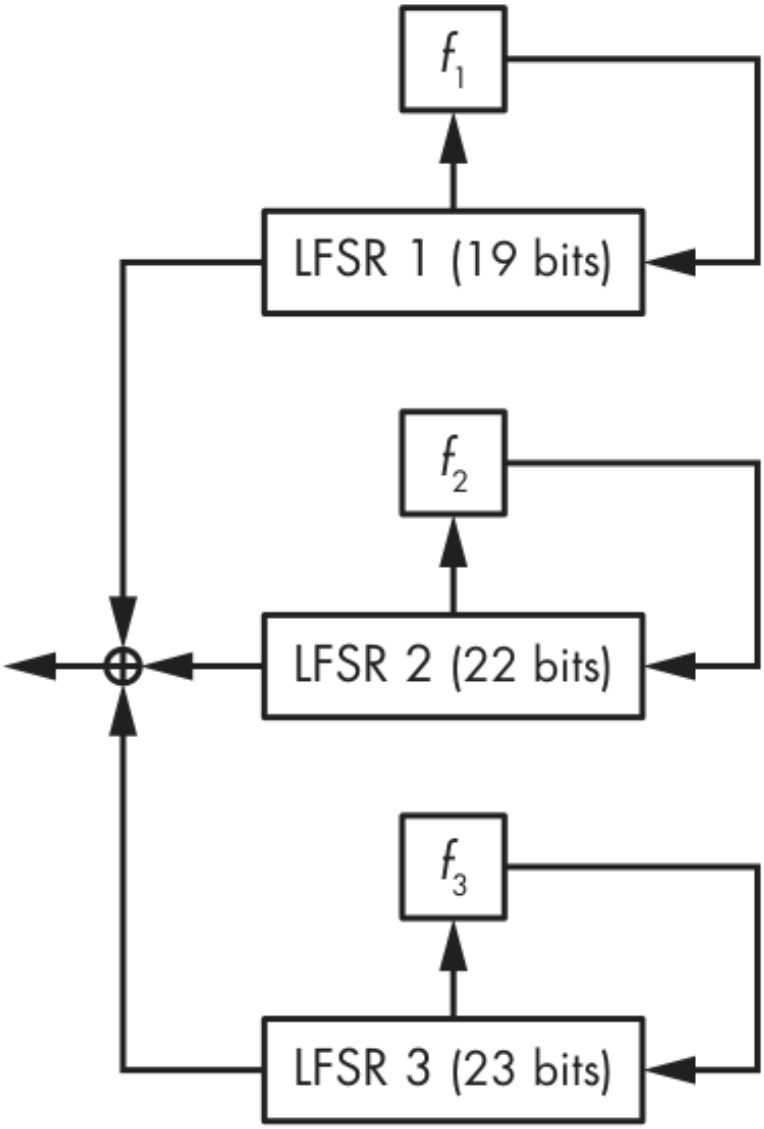

Рисунок 5-9. Шифр A5/1

В A5/1 применяются LFSR длиной 19, 22 и 23 бита со следующими многочленами:

```
1+X^14+X^17+X^18+X^19,
1+X^21+X^22,
1+X^8+X^21+X^22+X^23.
```

Как такая конструкция могла считаться безопасной, если в ней есть только LFSR и нет NFSR? Секрет заключается в механизме обновления A5/1. Вместо обновления всех трех LFSR на каждом такте разработчики A5/1 ввели следующее правило тактирования:

1. Проверяется значение 9-го бита LFSR 1, 11-го бита LFSR 2 и 11-го бита LFSR 3. Эти биты называются тактирующими. Среди этих трех битов либо все имеют одинаковое значение - 1 или 0, - либо два имеют одинаковое значение.
2. Тактируются те регистры, тактирующие биты которых равны значению большинства, 0 или 1. При каждом обновлении тактируются два либо три LFSR.

Без этого правила A5/1 вообще не обеспечивал бы никакой безопасности, и для взлома шифра достаточно обойти это правило. Однако сделать это труднее, чем сказать, в чем вы сейчас

убедитесь.

> **Примечание.** При нерегулярном правиле тактирования A5/1 каждый регистр тактируется при любом обновлении с вероятностью 3/4. Вероятность того, что хотя бы один из двух других регистров имеет то же значение бита, равна 1-(1/2)^2, где (1/2)^2 - вероятность того, что оба других регистра имеют отличающееся значение бита.

В связи 2G A5/1 используется с 64-битным ключом и 22-битным одноразовым номером, который меняется для каждого нового кадра данных. Механизм инициализации A5/1 сначала обнуляет все регистры, затем побитово вводит в каждый регистр ключ, а вслед за ним одноразовый номер; после ввода каждого бита регистры обновляются. После этого система еще 100 раз обновляется по описанному выше нерегулярному правилу.

Атаки на A5/1 восстанавливают 64-битное начальное состояние системы - совокупное начальное значение LFSR длиной 19, 22 и 23 бита. Затем, обращая механизм инициализации, они раскрывают одноразовый номер, если он еще не был известен, и ключ. Эти атаки являются атаками с известным открытым текстом (known-plaintext attacks, KPA), поскольку часть зашифрованных данных известна, что позволяет атакующим определить соответствующие участки ключевого потока, складывая посредством XOR шифртекст с известными фрагментами открытого текста.

Существуют два основных типа атак на A5/1:

Тонкие атаки. Используют внутреннюю линейность A5/1 и простую систему нерегулярного тактирования.

Грубые атаки. Используют только короткий ключ A5/1 и обратимость процедуры ввода номера кадра.

Посмотрим, как работают эти атаки.

Тонкие атаки

Мы рассмотрим тонкую атаку типа «угадай и определи». В такой атаке злоумышленник угадывает некоторые секретные значения состояния, чтобы определить остальные. В криптоанализе «угадывать» означает применять полный перебор: для каждого возможного значения LFSR 1, LFSR 2 и всех возможных значений тактирующего бита LFSR 3 в течение первых 11 тактов атака восстанавливает биты LFSR 3, решая уравнения, зависящие от угаданных битов. Когда догадка верна, атакующий получает правильное значение LFSR 3.

Псевдокод атаки выглядит следующим образом:

```
Для каждого из 2^19 значений начального состояния LFSR 1
    Для каждого из 2^22 значений начального состояния LFSR 2
        Для каждого из 2^11 значений тактирующего бита LFSR 3 в первые 11 тактов
            Восстановить начальное состояние LFSR 3
            Проверить догадку; если она верна, вернуть результат; иначе продолжить
```

Насколько эффективна эта атака по сравнению с поиском полным перебором за 2^64 испытаний, рассмотренным в главе 3? В худшем случае, когда алгоритм добивается успеха лишь при самой последней проверке, эта атака выполняет не более 2^19*2^22*2^11=2^52 операций. Это в 2^12, то есть примерно в 4000 раз, быстрее полного перебора, если предположить, что последние две операции в приведенном псевдокоде требуют примерно столько же вычислений, сколько проверка 64-битного ключа при полном переборе. Но верно ли это предположение?

Вспомните наше обсуждение полной стоимости атаки в главе 3. Оценивая стоимость атаки, нужно учитывать не только объем вычислений, но и возможность распараллеливания и расход памяти. Ни то ни другое здесь не представляет проблемы: как и любой полный перебор, атака «угадай и

определи» исключительно хорошо распараллеливается - при выполнении на N ядрах она становится в N раз быстрее - и требует не больше памяти, чем сам шифр.

Наша оценка стоимости атаки как 2^52 неточна и по другой причине. На самом деле каждая из 2^52 операций, то есть проверка одного кандидата на роль ключа, занимает примерно в четыре раза больше тактов, чем проверка ключа при полном переборе. Поэтому в пересчете на полный перебор реальная стоимость этой конкретной атаки ближе к 4*2^52=2^54 операций.

Атака «угадай и определи» на A5/1 позволяет расшифровывать защищенную мобильную связь, однако при выполнении на кластере специализированных аппаратных устройств восстановление ключа занимает пару часов. Иными словами, до расшифровки в реальном времени ей далеко. Для этого существует атака другого типа.

Грубые атаки

Грубая атака на A5/1 основана на компромиссе между временем и памятью (time-memory trade-off, TMTO). Ей не важно внутреннее устройство A5/1; важно лишь то, что его состояние имеет длину 64 бита. TMTO-атака рассматривает A5/1 как черный ящик, который принимает 64-битное значение состояния и выдает 64-битное значение - первые 64 бита ключевого потока.

Идея атаки состоит в том, чтобы снизить стоимость полного перебора за счет использования большого объема памяти. Простейшая разновидность TMTO представляет собой атаку по кодовой книге: заранее вычисляется таблица из 2^64 элементов, содержащая пары «ключ:значение», и для каждого из 2^64 возможных ключей сохраняется выходное значение. Чтобы использовать эту заранее вычисленную таблицу для атаки, достаточно получить выходные данные экземпляра A5/1, а затем найти в таблице соответствующий им ключ. Сама атака выполняется быстро - требуется лишь время на поиск значения в памяти, - однако построение таблицы требует 2^64 вычислений A5/1. Хуже того, атакам по кодовой книге нужен безумный объем памяти: 2^64*(64+64) бит, то есть 2^68 байт, или 256 эксабайт. Это десятки центров обработки данных, так что о таком варианте можно забыть.

TMTO-атаки снижают требования атак по кодовой книге к памяти ценой увеличения объема вычислений на оперативной стадии атаки. Чем меньше таблица, тем больше вычислений требуется для взлома ключа. В любом случае подготовка таблицы стоит около 2^64 операций, но выполнить ее нужно лишь один раз.

В 2010 году исследователи примерно за два месяца создали таблицы общим объемом два терабайта, используя графические процессоры (GPU) и запуская 100 000 экземпляров A5/1 параллельно. С помощью столь больших таблиц разговоры, зашифрованные A5/1, можно было расшифровывать почти в реальном времени. Операторы связи внедрили обходные меры для ослабления атаки, но настоящее решение появилось в стандартах мобильной связи 3G и 4G, где от A5/1 полностью отказались.

Потоковые шифры, ориентированные на программную реализацию

Программные потоковые шифры работают с байтами или 32- либо 64-битными словами, а не с отдельными битами. Это эффективнее на современных центральных процессорах, где инструкции могут выполнять арифметические операции над словом за то же время, что и над одним битом. Поэтому программные потоковые шифры лучше подходят, чем аппаратные, для серверов и браузеров, работающих на персональных компьютерах, где мощные процессоры общего назначения выполняют шифр как обычную программу.

Сегодня программные потоковые шифры вызывают значительный интерес по нескольким причинам. Во-первых, поскольку многие устройства оснащены мощными CPU, а аппаратное обеспечение подешевело, потребность в маленьких побитовых шифрах снизилась. Например, два потоковых шифра стандарта мобильной связи 4G - европейский SNOW3G и китайский ZUC - в отличие от более старого A5/1 работают с 32-битными словами, а не с битами.

Во-вторых, потоковые шифры стали популярнее в программном обеспечении за счет блочных шифров, особенно после фиаско с атакой через оракул дополнения на блочные шифры в режиме CBC. Кроме того, потоковые шифры проще специфицировать и реализовать, чем блочные: вместо смешивания битов сообщения и ключа потоковый шифр просто принимает биты ключа как секрет. В действительности один из самых популярных потоковых шифров - это замаскированный блочный шифр: AES в режиме счетчика (CTR).

Одна из конструкций программного потокового шифра, применяемая в SNOW3G и ZUC, копирует аппаратные шифры и их FSR, заменяя биты байтами или словами. Однако для криптографа это не самые интересные конструкции. На момент написания книги наибольший интерес представляют две конструкции - RC4 и Salsa20, используемые во множестве систем, несмотря на то что одна из них полностью взломана.

RC4

RC4 был разработан в 1987 году Роном Ривестом из RSA Security, а затем восстановлен методами обратной разработки и опубликован без разрешения в 1994 году. Долгое время он оставался самым широко применяемым потоковым шифром. RC4 использовался в бесчисленном множестве приложений, наиболее известные из которых - 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, рассмотрим работу RC4.

Как работает RC4

RC4 - один из самых простых когда-либо созданных шифров. Он не выполняет никаких похожих на криптографию операций: никаких XOR, умножений, S-блоков... ничего. Он просто переставляет байты. Внутреннее состояние RC4 представляет собой массив `S` из 256 байтов, которому сначала присваиваются значения `S[0]=0`, `S[1]=1`, `S[2]=2`, ..., `S[255]=255`, а затем он инициализируется `n`-байтовым ключом `K` с помощью алгоритма планирования ключей (key scheduling algorithm, KSA), показанного на Python в листинге 5-1.

```
j = 0
# Задать S как массив S[0] = 0, S[1] = 1, ..., S[255] = 255.
S = range(256)
# Перебрать i от 0 до 255.
for i in range(256):
    # Вычислить сумму v.
    j = (j + S[i] + K[i % n]) % 256
    # Поменять местами S[i] и S[j].
    S[i], S[j] = S[j], S[i]
```

Листинг 5-1. Алгоритм планирования ключей RC4

После завершения этого алгоритма массив `S` по-прежнему содержит все значения байтов от 0 до 255, но теперь они расположены в кажущемся случайным порядке. Например, при 128-битном ключе из одних нулей состояние `S`, от `S[0]` до `S[255]`, принимает вид:

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

Однако если я инвертирую первый бит ключа и снова выполню KSA, то получу совершенно другое, по видимости случайное состояние:

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

Имея начальное состояние `S`, RC4 генерирует ключевой поток `KS` той же длины, что и открытый текст `P`, и вычисляет шифртекст как `C=P xor KS`. Если длина `P` равна `m` байтам, байты ключевого потока `KS` вычисляются из `S` приведенным в листинге 5-2 кодом на Python.

```
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]
```

Листинг 5-2. Генерация ключевого потока RC4, где `S` - состояние, инициализированное в листинге 5-1

В листинге 5-2 каждая итерация цикла `for` изменяет не более двух байтов внутреннего состояния `S` шифра RC4: `S[i]` и `S[j]`, значения которых меняются местами. Иными словами, если `i=0`, `j=4`, `S[0]=56` и `S[4]=78`, то операция перестановки присваивает `S[0]` значение 78, а `S[4]` - значение 56. Если `j` равно `i`, то `S[i]` не изменяется.

Это выглядит слишком просто, чтобы быть безопасным, и тем не менее криптоаналитикам потребовалось 20 лет, чтобы найти пригодные для эксплуатации недостатки. До их обнаружения слабости RC4 были известны лишь в отдельных реализациях, например в первом стандарте шифрования Wi-Fi - WEP.

RC4 в WEP

WEP, протокол безопасности Wi-Fi первого поколения, теперь полностью взломан из-за слабостей как в конструкции самого протокола, так и в RC4.

В реализации WEP RC4 шифрует полезные данные кадров 802.11 - дейтаграмм, или пакетов, передающих данные по беспроводной сети. Для всех полезных данных в одном сеансе используется один и тот же секретный ключ длиной 40 или 104 бита, но при этом в заголовке кадра, то есть предшествующей полезным данным части с метаданными, кодируется якобы уникальный трехбайтовый одноразовый номер.

Проблема в том, что RC4 не поддерживает одноразовый номер, по крайней мере в официальной спецификации, а использовать потоковый шифр без одноразового номера нельзя. Разработчики WEP обошли это ограничение: включили 24-битный одноразовый номер в заголовок беспроводного

кадра и приписали его перед ключом WEP, получив секретный ключ RC4. Иными словами, если одноразовый номер состоит из байтов `N[0]`, `N[1]`, `N[2]`, а ключ WEP - из `K[0]`, `K[1]`, `K[2]`, `K[3]`, `K[4]`, фактическим ключом RC4 становится `N[0]`, `N[1]`, `N[2]`, `K[0]`, `K[1]`, `K[2]`, `K[3]`, `K[4]`. В результате 40-битные секретные ключи дают эффективные 64-битные ключи, а 104-битные - эффективные 128-битные. Что получается? Рекламируемый как 128-битный протокол WEP в лучшем случае обеспечивает лишь 104-битную безопасность.

Но настоящие проблемы с приемом WEP для одноразовых номеров заключаются в следующем:

Одноразовые номера слишком малы - всего 24 бита. Если для каждого нового сообщения выбирать одноразовый номер случайно, придется дождаться примерно `2^(24/2)=2^12` пакетов, то есть нескольких мегабайтов трафика, прежде чем найдутся два пакета, зашифрованных с одним одноразовым номером, а значит, с одним ключевым потоком. Даже если одноразовый номер является счетчиком от 0 до `2^24-1`, до переполнения потребуется передать несколько гигабайтов данных; после повторения одноразового номера атакующий сможет расшифровывать пакеты. Но есть проблема серьезнее.

Такое объединение одноразового номера и ключа помогает восстановить ключ. Три несекретных байта одноразового номера WEP позволяют атакующему определить значение `S` после трех итераций алгоритма планирования ключей. Благодаря этому криптоаналитики выяснили, что первый байт ключевого потока сильно зависит от первого секретного байта ключа - четвертого байта, принятого KSA, - и что это смещение можно использовать для восстановления секретного ключа.

Для эксплуатации этих слабостей нужен доступ и к шифртексту, и к ключевому потоку, то есть известный или выбранный открытый текст. Но получить их достаточно легко: известный открытый текст встречается, когда кадры Wi-Fi инкапсулируют данные с известным заголовком, а выбранный открытый текст - когда атакующий внедряет известный текст, зашифрованный целевым ключом. В итоге атаки работают на практике, а не только на бумаге.

После появления первых атак на WEP в 2001 году исследователи нашли более быстрые атаки, которым требовалось меньше шифртекстов. Сегодня существуют даже такие инструменты, как aircrack-ng, реализующие атаку целиком - от перехвата сетевого трафика до криптоанализа.

Небезопасность WEP обусловлена как слабостями RC4, который принимает одноразовый ключ вместо ключа и одноразового номера, как любой достойный потоковый шифр, так и слабостями конструкции самого WEP.

Теперь рассмотрим второй по значимости провал RC4.

RC4 в TLS

TLS - самый важный протокол безопасности, используемый в интернете. В первую очередь он известен как основа HTTPS-соединений, но применяется также для защиты некоторых соединений виртуальных частных сетей (VPN), почтовых серверов, мобильных приложений и многого другого. И, к сожалению, TLS долгое время поддерживал RC4.

В отличие от WEP, реализация TLS не совершает той же вопиющей ошибки, изменяя спецификацию RC4 ради применения открытого одноразового номера. Вместо этого TLS просто передает RC4 уникальный 128-битный сеансовый ключ, поэтому он сломан несколько меньше, чем WEP.

Слабость TLS обусловлена исключительно самим RC4 и его непростительными дефектами: статистическими смещениями, то есть неслучайностью, которая, как мы знаем, совершенно неприемлема для потокового шифра. Например, второй байт ключевого потока, создаваемого RC4,

равен нулю с вероятностью 1/128, тогда как в идеале она должна быть равна 1/256. Напомним, что байт может принимать 256 значений от 0 до 255, поэтому действительно случайный байт равен нулю с вероятностью 1/256. Еще удивительнее, что большинство специалистов продолжали доверять RC4 вплоть до 2013 года, хотя о его статистических смещениях было известно с 2001 года.

Одних известных статистических смещений RC4 должно было хватить, чтобы полностью отказаться от шифра, даже если мы еще не знали, как использовать эти смещения для компрометации реальных приложений. Публично недостатки RC4 в TLS не эксплуатировались до 2011 года, однако предположительно АНБ сумело использовать слабости RC4 для компрометации TLS-соединений с RC4 намного раньше.

Как выяснилось, смещен был не только второй байт ключевого потока RC4, но и все первые 256 байтов. В 2011 году исследователи обнаружили, что вероятность равенства одного из этих байтов нулю составляет 1/256+c/256^2 для некоторой константы c, принимающей значения от 0,24 до 1,34. Это относится не только к нулевому, но и к другим значениям байта. Удивительно, что RC4 терпит неудачу там, где справляются даже многие некриптографические PRNG: при формировании равномерно распределенных псевдослучайных байтов, когда каждое из 256 значений появляется с вероятностью 1/256.

Ошибочную реализацию RC4 в TLS можно эксплуатировать даже в самой слабой модели атаки - с выбранным шифртекстом: вы собираете шифртексты и ищете открытый текст, а не ключ. Но есть одно условие: потребуется много шифртекстов, в которых один и тот же открытый текст несколько раз зашифрован разными секретными ключами. Иногда такую модель называют широковещательной, поскольку она напоминает передачу одного сообщения множеству получателей.

Предположим, требуется расшифровать байт P_1 открытого текста P, имея множество байтов шифртекста, полученных при перехвате разных шифртекстов одного сообщения. Для четырех ключевых потоков KS^1,...,KS^4 мы получим первые байты каждого из четырех шифртекстов C^1,...,C^4, такие что

```
C_1^1 =P_1 xor KS_1^1,
C_1^2 =P_1 xor KS_1^2,
C_1^3 =P_1 xor KS_1^3,
C_1^4 =P_1 xor KS_1^4.
```

Из-за смещения RC4 байты ключевого потока KS_1^i, то есть первый байт в каждом из четырех экземпляров, чаще равны нулю, чем любому другому значению. Поэтому байты C_1^i чаще равны P_1, чем любому другому значению. Чтобы определить P_1 по байтам C_1^i, достаточно подсчитать число появлений каждого значения байта и принять за P_1 самое частое. Однако статистическое смещение очень мало, поэтому для сколько-нибудь надежного результата нужны миллионы значений.

Атаку можно обобщить так, чтобы восстанавливать более одного байта открытого текста и использовать более одного смещенного значения, в данном случае нуля. Алгоритм лишь немного усложнится. Однако такую атаку трудно осуществить на практике, потому что требуется собрать множество шифртекстов, содержащих один открытый текст, но полученных с разными ключами. Например, эта атака не сможет взломать все защищенные TLS соединения, использующие RC4, поскольку сервер нужно заставить зашифровать один и тот же открытый текст для множества разных получателей либо многократно для одного получателя, но с разными ключами.

Salsa20

Salsa20 - простой программно-ориентированный шифр, оптимизированный для современных CPU и реализованный во множестве протоколов и библиотек вместе со своим вариантом ChaCha. Его разработчик, уважаемый криптограф Дэниел Дж. Бернстайн, представил Salsa20 на конкурс eSTREAM в 2005 году, и шифр вошел в программный портфель eSTREAM. Простота и скорость Salsa20 сделали его популярным среди разработчиков.

Salsa20 - потоковый шифр на основе счетчика: он генерирует ключевой поток, многократно обрабатывая счетчик, увеличиваемый для каждого блока. Как показано на рисунке 5-10, основной алгоритм Salsa20 преобразует 512-битный блок, используя ключ `K`, одноразовый номер `N` и значение счетчика `Ctr`. Затем Salsa20 складывает результат с исходным значением блока и получает блок ключевого потока. Если бы алгоритм непосредственно возвращал результат перестановки ядра, Salsa20 был бы полностью небезопасен, поскольку эту перестановку можно обратить. Финальное сложение исходного секретного состояния `K || N || Ctr` делает преобразование из ключа в блок ключевого потока необратимым.

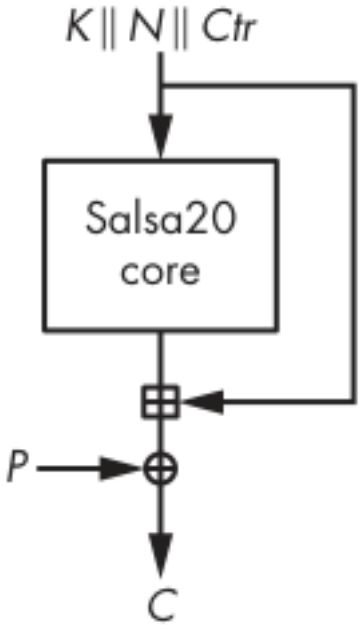

Рисунок 5-10. Схема шифрования Salsa20 для 512-битного блока открытого текста

Использование функции четверть-раунда

Основная перестановка Salsa20 использует функцию, называемую четверть-раундом (quarter-round, QR), которая преобразует четыре 32-битных слова `a`, `b`, `c` и `d` следующим образом:

```
b =b xor [(a+d)<<<7],
c =c xor [(b+a)<<<9],
d =d xor [(c+b)<<<13],
a =a xor [(d+c)<<<18].
```

Эти четыре строки вычисляются сверху вниз. Поэтому новое значение `b` зависит от `a` и `d`, новое значение `c` зависит от `a` и нового значения `b`, а значит, также от `d`, и так далее.

Операция <<< обозначает циклический сдвиг слова влево на указанное число битов, которое для 32-битных слов может принимать любое значение от 1 до 31. Например, <<< 8 циклически сдвигает биты слова на восемь позиций влево, как показывают следующие примеры:

```
0x01234567 <<<  8 = 0x23456701
0x01234567 <<< 16 = 0x45670123
0x01234567 <<< 22 = 0x59c048d1
```

Преобразование 512-битного состояния Salsa20

Основная перестановка Salsa20 преобразует 512-битное внутреннее состояние, рассматриваемое как массив 4*4 из 32-битных слов. На рисунке 5-11 показано начальное состояние, включающее ключ из восьми слов, или 256 битов, одноразовый номер из двух слов, или 64 битов, счетчик из двух слов, или 64 битов, и четыре фиксированных слова-константы, или 128 битов, одинаковых для каждого шифрования и расшифрования и для всех блоков.

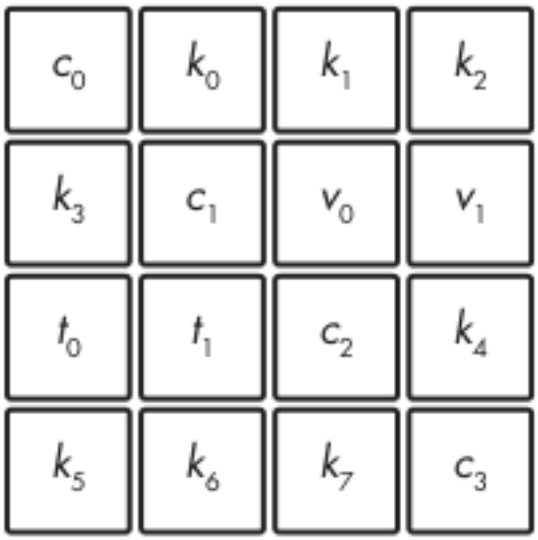

Рисунок 5-11. Инициализация состояния Salsa20

Чтобы преобразовать исходное 512-битное состояние, Salsa20 сначала независимо применяет преобразование QR ко всем четырем столбцам - это называется раундом столбцов, - а затем независимо ко всем четырем строкам - раундом строк, - как показано на рисунке 5-12. Последовательность «раунд столбцов/раунд строк» представляет собой двойной раунд. Salsa20 повторяет 10 двойных раундов, то есть всего 20 раундов; именно поэтому в названии Salsa20 присутствует число 20.

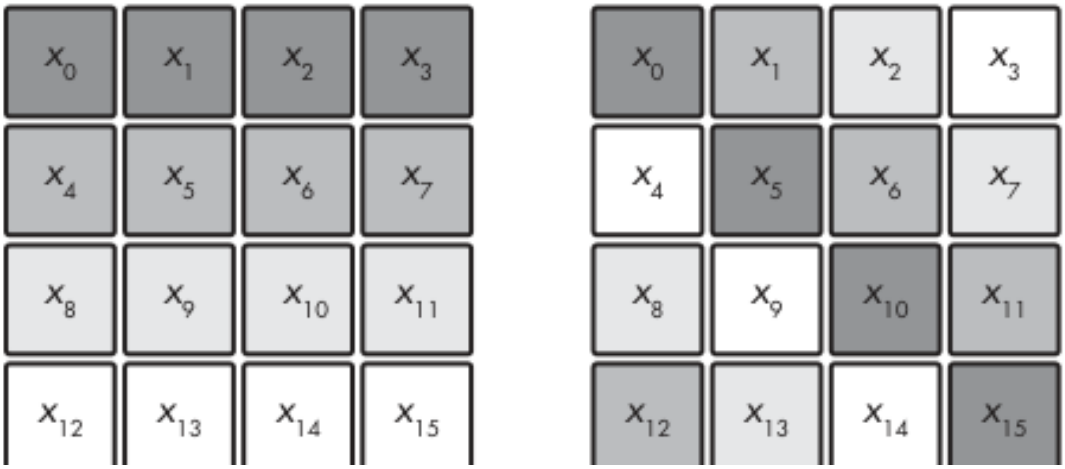

Рисунок 5-12. Столбцы и строки, преобразуемые функцией четверть-раунда (QR) Salsa20

Раунд столбцов преобразует четыре столбца следующим образом:

```
QR(x_0,x_4,x_8,x_12),
QR(x_1,x_5,x_9,x_13),
QR(x_2,x_6,x_10,x_14),
QR(x_3,x_7,x_11,x_15).
```

Раунд строк преобразует строки следующим образом:

```
QR(x_0,x_1,x_2,x_3),
QR(x_5,x_6,x_7,x_4),
QR(x_10,x_11,x_8,x_9),
QR(x_15,x_12,x_13,x_14).
```

В раунде столбцов каждая функция QR принимает аргументы `x_i`, упорядоченные сверху вниз, а в раунде строк первым аргументом QR служат слова на диагонали, как в правом массиве на рисунке 5-12, а не слова первого столбца.

Оценка Salsa20

В листинге 5-3 показаны начальные состояния Salsa20 для первого и второго блоков при инициализации ключом из одних нулевых байтов `00` и одноразовым номером из одних единичных байтов `ff`. Эти два состояния отличаются лишь одним битом счетчика, выделенным полужирным в оригинале: для первого блока он равен 0, а для второго - 1.

```
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
```

Листинг 5-3. Начальные состояния Salsa20 для первых двух блоков при ключе из одних нулей и одноразовом номере из одних единиц

И все же, несмотря на различие всего в один бит, после 10 двойных раундов соответствующие внутренние состояния совершенно не похожи друг на друга, как показано в листинге 5-4.

```
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
```

Листинг 5-4. Состояния из листинга 5-3 после 10 двойных раундов Salsa20

Но помните: даже если значения слов в блоке ключевого потока выглядят случайными, это далеко не гарантия безопасности. Выход RC4 выглядит случайным, однако имеет явные смещения. К

счастью, Salsa20 намного безопаснее RC4 и не имеет статистических смещений. Тем не менее даже статистическая неотличимость ключевых потоков от идеально случайных байтов недостаточна для достижения криптографической безопасности.

Знакомство с дифференциальным криптоанализом

Чтобы показать, почему Salsa20 безопаснее RC4, рассмотрим основы дифференциального криптоанализа - исследования различий между состояниями, а не самих их значений. Например, два начальных состояния в листинге 5-3 отличаются одним битом счетчика, то есть словом x_8 массива состояния Salsa20. Следующий массив показывает побитовую разность этих двух состояний:

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

В действительности разность двух состояний представляет собой XOR этих состояний. Единичный бит соответствует различию двух состояний в одном бите. В результате XOR двух состояний любые ненулевые биты указывают на различия.

Чтобы увидеть, как быстро изменения исходного состояния распространяются под действием основного алгоритма Salsa20, проследим разность двух состояний по мере выполнения раундов. После одного раунда различие распространяется по первому столбцу на два из трех остальных слов этого столбца:

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

После двух раундов различия продолжают распространяться по строкам, в которых уже есть различие, то есть по всем, кроме второй. На этом этапе разности состояний остаются довольно разреженными: внутри каждого слова изменилось не так много битов.

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

После трех раундов различия между состояниями становятся плотнее, хотя множество нулевых полубайтов показывает, что исходное различие все еще не затронуло многие позиции битов:

```
3ab3c25d 9f40a5c9 10070e30 07bd03c0
db1ee2ce 43ee9401 21a7022c 348fd80c
403c1e72 00034003 4dc843be 700b8857
5625b75b 09c00e00 06000348 23f712d4
```

После четырех раундов различия выглядят случайными для наблюдателя-человека и почти случайны статистически:

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

Всего после четырех раундов единичное различие распространяется на большинство битов 512-битного состояния. В криптографии это называется полной диффузией.

Различия не просто распространяются по всему состоянию: они делают это в соответствии со сложными уравнениями, из-за которых будущие различия трудно предсказать. Эволюцией состояния управляют высоконелинейные отношения благодаря сочетанию XOR, сложения и циклического сдвига. Если бы использовались только XOR, множество различий по-прежнему

распространялось бы, но процесс был бы линейным и потому небезопасным.

Атака на Salsa20/8

По умолчанию Salsa20 выполняет 20 раундов, но для повышения скорости иногда применяется версия всего с 12 раундами, называемая Salsa20/12. Хотя Salsa20/12 использует на восемь раундов меньше, чем Salsa20, согласно современному состоянию исследований на практике она столь же надежна, как и 20-раундовая версия. Даже Salsa20/8 всего с восемью раундами считается слабее лишь теоретически, а на практике столь же надежна, как Salsa20.

В идеале взлом Salsa20 должен требовать `2^256` операций благодаря 256-битному ключу. Если ключ можно восстановить менее чем за `2^256` операций, теоретически шифр взломан. Именно так обстоит дело с Salsa20/8.

Атака на Salsa20/8, опубликованная в статье 2008 года «New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba», одним из соавторов которой был я и за которую мы получили от Дэниела Дж. Бернстайна премию по криптоанализу, использует статистическое смещение в основном алгоритме Salsa после четырех раундов, чтобы восстановить ключ восьмираундовой Salsa20. В действительности это преимущественно теоретическая атака: мы оцениваем ее сложность в `2^251` операций основной функции. Это неосуществимо, как и любое вычисление порядка `2^100` операций или более, но все же меньше ожидаемой сложности взлома `2^256`.

Атака использует не только смещение в первых четырех раундах Salsa20/8, но и свойство последних четырех раундов: если известны одноразовый номер `N` и счетчик `Ctr`, показанные на рисунке 5-10, единственным значением, необходимым для обращения вычисления от ключевого потока к начальному состоянию, остается ключ `K`. Но, как показано на рисунке 5-13, зная лишь некоторую часть `K`, можно частично обратить вычисление до четвертого раунда и наблюдать некоторые биты этого промежуточного состояния, включая смещенный бит. Смещение наблюдается только при правильной догадке о части ключа, поэтому оно служит признаком правильного ключа.

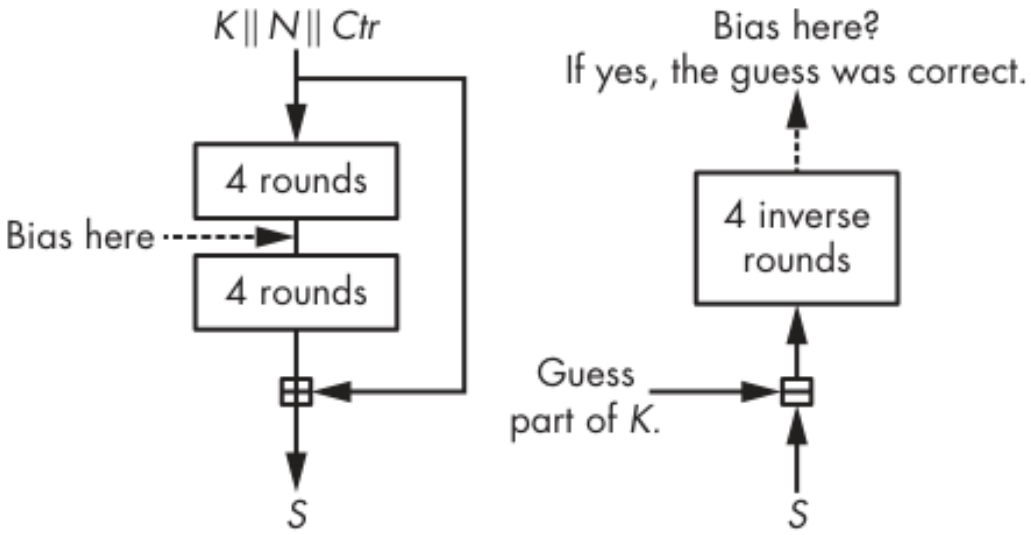

Рисунок 5-13. Принцип атаки на Salsa20/8

В реальной атаке на Salsa20/8 для определения правильной догадки требуется угадать 220 битов ключа и получить `2^31` пар блоков ключевого потока с одной и той же конкретной разностью одноразовых номеров. После нахождения правильных 220 битов оставшиеся 36 битов перебираются полностью. Полный перебор требует `2^36` операций, что ничтожно по сравнению с нереалистичными `2^220*2^31=2^251` испытаниями, необходимыми для нахождения 220 битов на первом этапе атаки.

Что может пойти не так

Увы, с потоковыми шифрами многое может пойти не так: от хрупких, небезопасных конструкций до неправильных реализаций стойких алгоритмов. В следующих разделах я рассмотрю каждую категорию возможных проблем.

Повторное использование одноразового номера

Самая распространенная ошибка при работе с потоковыми шифрами - многократное использование одноразового номера с одним и тем же ключом. При этом получаются одинаковые ключевые потоки, что позволяет взломать шифрование. Например, если сложить два шифртекста посредством XOR, ключевой поток исчезнет и останется XOR двух открытых текстов.

Реальный пример - старые версии Microsoft Word и Excel, использовавшие уникальный одноразовый номер для каждого документа, но не менявшие его при изменении документа. В результате открытый и зашифрованный тексты старой версии документа позволяли расшифровать более поздние зашифрованные версии. Если такую ошибку допустила Microsoft, можно представить масштаб этой проблемы.

Некоторые потоковые шифры, разработанные в 2010-х годах, пытались снизить риск повторного использования одноразовых номеров при помощи конструкций, «устойчивых к неправильному применению», то есть шифров, сохраняющих безопасность даже при двукратном использовании одноразового номера. Однако такой уровень безопасности снижает производительность, как вы увидите в главе 8 при рассмотрении режима SIV.

Ошибочная реализация RC4

Хотя RC4 слаб сам по себе, слепая оптимизация реализации может сделать его еще слабее. В качестве примера рассмотрим одну работу с конкурса Underhanded C Contest 2007 года - неофициального состязания, в котором программисты пишут выглядящий безобидным код, на самом деле содержащий вредоносную функцию.

Вот как это работает. Наивная реализация строки `swap(S[i], S[j])` в алгоритме RC4 выглядит следующим образом, как показывает этот код на Python:

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

Такой способ перестановки двух переменных работает, но требует создания новой переменной `buf`. Чтобы обойтись без нее, программисты часто используют следующий прием XOR-перестановки значений переменных x и y:

```
x = x XOR y
y = x XOR y
x = x XOR y
```

Он работает потому, что вторая строка присваивает y значение `x xor y xor y=x`, а третья присваивает x значение `x xor y xor x xor y xor y=y`. Применение этого приема для реализации RC4 дает код листинга 5-5, адаптированный из программы Дэвида Вагнера и Филиппа Бионди, представленной на Underhanded C Contest 2007 года и доступной по адресу http://www.underhanded-c.org/_page_id_16.html.

```
#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];
}
```

Листинг 5-5. Неправильная реализация RC4 на C из-за применения XOR-перестановки

Можете ли вы заметить проблему с XOR-перестановкой?

Все идет наперекосяк, когда `i=j`. Вместо сохранения состояния без изменений XOR-перестановка присваивает `S[i]` значение `S[i] xor S[i]=0`. В результате каждый раз, когда `i` становится равно `j` при планировании ключа или шифровании, один байт состояния обнуляется; в конце концов все состояние, а значит, и весь ключевой поток становятся нулевыми. Например, после обработки 68 Кбайт данных большинство байтов 256-байтового состояния равны нулю, и выходной ключевой поток выглядит так:

```
00 00 00 00 00 00 00 53 53 00 00 00 00 00 00 00 00 00 00 13 13 00 5c 00 a5 00 00 ...
```

Урок заключается в том, что не следует чрезмерно оптимизировать криптографические реализации. В криптографии ясность и уверенность всегда важнее производительности.

Слабые шифры, встроенные в аппаратуру

Когда криптосистема оказывается небезопасной, одни системы могут быстро и незаметно обновить затронутое программное обеспечение удаленно, как веб-приложения, либо выпустить новую версию и предложить пользователям обновиться, как мобильные приложения. Другим системам везет меньше, и до перехода на безопасную версию им некоторое время приходится сохранять скомпрометированную криптосистему; так обстоит дело с некоторыми спутниковыми телефонами.

В начале 2000-х годов американский и европейский институты стандартизации телекоммуникаций TIA и ETSI совместно разработали два стандарта связи для спутниковых телефонов. Спутниковый телефон похож на мобильный, но его сигнал проходит через спутники, а не через наземные станции. Преимущество состоит в том, что при наличии спутникового покрытия им можно пользоваться практически в любой точке мира. Недостатки - цена, качество, задержка и, как оказалось, безопасность.

GMR-1 и GMR-2 - два стандарта спутниковой телефонии, принятые большинством коммерческих поставщиков, например Thuraya и Inmarsat. Оба включают потоковые шифры для шифрования голосовой связи. Шифр GMR-1 ориентирован на аппаратную реализацию и объединяет четыре LFSR, подобно A5/2 - намеренно небезопасному шифру стандарта мобильной связи 2G, предназначенному для незападных стран. Шифр GMR-2 ориентирован на программную реализацию, имеет восьмибайтовое состояние и использует S-блоки. Оба потоковых шифра небезопасны и защищают пользователей только от любителей, но не от государственных ведомств.

Эта история напоминает нам, что потоковые шифры прежде было легче взломать, чем блочные, и их легче саботировать. Почему? Если намеренно разработать слабый потоковый шифр, после обнаружения изъяна всегда можно сослаться на слабость потоковых шифров и отрицать злой умысел.

Дополнительная литература

Чтобы больше узнать о потоковых шифрах, начните с архивов конкурса eSTREAM по адресу https://www.ecrypt.eu.org/stream/project.html. Там находятся сотни статей о потоковых шифрах, включая подробные сведения более чем о 30 кандидатах и множестве атак. Среди самых интересных атак - корреляционные, алгебраические и кубические. Для первых двух типов особенно стоит обратиться к работам Николя Куртуа и Вилли Майера, а для кубических атак - к работам Итаи Динура и Ади Шамира.

Чтобы больше узнать об атаках на RC4, найдите атаку Скотта Флюрера, Ицика Мантина и Ади Шамира (FMS) 2001 года и исследовательскую статью 2013 года «On the Security of RC4 in TLS».

Наследие Salsa20 тоже заслуживает внимания. Потоковый шифр ChaCha похож на Salsa20, но использует немного другую основную перестановку, впоследствии примененную в хеш-функции BLAKE, как вы увидите в главе 6. Все эти алгоритмы используют методы программной реализации Salsa20 с параллельными инструкциями, рассмотренные на странице https://cr.yp.to/snuffle.html.

Глава 6. Хеш-функции

Хеш-функции, такие как SHA-256, SHA3 и BLAKE3, образуют швейцарский нож криптографа: они используются в цифровых подписях, шифровании с открытым ключом, проверке целостности, аутентификации сообщений, защите паролей, протоколах согласования ключей и многих других криптографических протоколах.

Шифруете ли вы электронное письмо, отправляете сообщение с мобильного телефона, подключаетесь к HTTPS-сайту или к удаленной машине через виртуальную частную сеть (VPN) либо Secure Shell (SSH), где-то внутри всегда работает хеш-функция.

Хеш-функции намного универсальнее и распространеннее всех прочих криптографических алгоритмов. Вот лишь некоторые области их применения: системы облачного хранения используют их для выявления одинаковых и измененных файлов; система управления версиями Git - для идентификации файлов в репозитории; системы обнаружения угроз на конечных устройствах и реагирования на них (endpoint detection and response, EDR) - для обнаружения измененных файлов; сетевые системы обнаружения вторжений (network-based intrusion detection systems, NIDS) используют хеши для обнаружения известных вредоносных данных, проходящих по сети; специалисты по компьютерной криминалистике используют значения хешей, чтобы доказать, что цифровые артефакты не были изменены; Bitcoin использует хеш-функцию в системе доказательства выполнения работы - и это далеко не все примеры.

В отличие от потоковых шифров, создающих длинный выход из короткого входа, хеш-функции принимают длинные входные данные и формируют короткий результат, называемый значением хеша или дайджестом, как показано на рисунке 6-1.

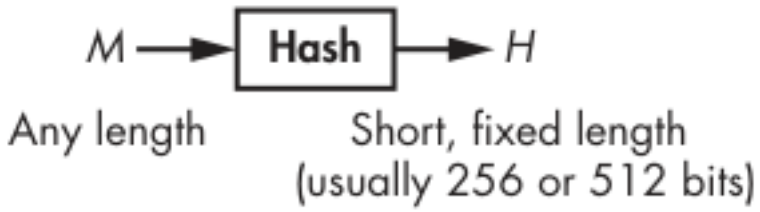

Рисунок 6-1. Вход и выход хеш-функции

Эта глава посвящена двум основным темам. Первая - безопасность: что значит «хеш-функция безопасна»? Для ответа я введу два важнейших понятия - устойчивость к коллизиям и устойчивость к нахождению прообраза. Вторая тема - построение хеш-функций. Я объясню общие методы, применяемые современными хеш-функциями, а затем рассмотрю внутреннее устройство самых распространенных функций: SHA-1, SHA-2, SHA-3 и BLAKE2. Наконец, вы увидите, как безопасные хеш-функции могут вести себя небезопасно при неправильном применении.

> **Примечание.** Не путайте криптографические хеш-функции с некриптографическими. Некриптографические хеш-функции применяются в таких структурах данных, как хеш-таблицы, или для обнаружения случайных ошибок и не обеспечивают никакой безопасности. Например, циклические избыточные коды (cyclic redundancy checks, CRC) - это некриптографические хеши, используемые для обнаружения случайных изменений файла.

Безопасные хеш-функции

Понятие безопасности хеш-функций отличается от того, что мы обсуждали до сих пор. Шифры защищают конфиденциальность данных, стремясь гарантировать, что данные, передаваемые в открытом виде, невозможно прочитать, а хеш-функции защищают целостность данных, стремясь гарантировать, что данные, передаваемые открыто или в зашифрованном виде, не были изменены. Если хеш-функция безопасна, два разных фрагмента данных всегда должны иметь разные хеши. Поэтому хеш файла может служить его идентификатором.

Рассмотрим самое распространенное применение хеш-функции - цифровые подписи, или просто подписи. При использовании цифровых подписей приложения обрабатывают хеш подписываемого сообщения, а не само сообщение, как показано на рисунке 6-2.

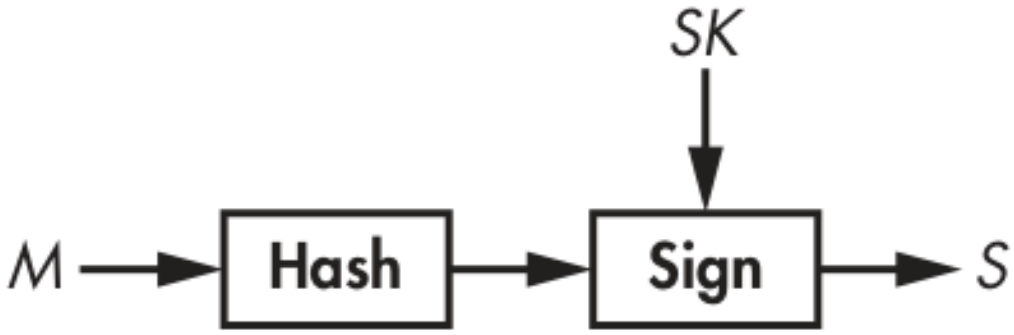

Рисунок 6-2. Хеш-функция в схеме цифровой подписи, где хеш служит заменителем сообщения

Хеш выступает идентификатором сообщения. Если в сообщении изменить хотя бы один бит, его хеш станет совершенно другим. Таким образом, хеш-функция помогает удостовериться, что сообщение не было изменено. Подписывать хеш сообщения так же безопасно, как само сообщение, а подписать короткий хеш длиной, например, 256 битов намного быстрее, чем потенциально очень большое сообщение. В действительности большинство алгоритмов подписи работает только с короткими входными данными, такими как значения хеша.

И снова непредсказуемость

Криптографическая стойкость хеш-функций проистекает из непредсказуемости их выходных данных. Рассмотрим следующие 256-битные шестнадцатеричные значения: эти хеши вычислены стандартной хеш-функцией NIST SHA-256 для входных данных в виде ASCII-букв `a`, `b` и `c`. Хотя значения `a`, `b` и `c` отличаются всего на 1 или 2 бита (`a` - это битовая последовательность `01100001`, `b` - `01100010`, а `c` - `01100011`), их значения хеша совершенно различны:

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

Имея только эти три хеша, невозможно предсказать значение хеша SHA-256 для `d` или хотя бы один его бит, поскольку значения безопасной хеш-функции непредсказуемы. Безопасная хеш-функция должна быть подобна черному ящику, который при получении входных данных каждый раз возвращает случайную строку.

Общее теоретическое определение безопасной хеш-функции требует, чтобы она вела себя как истинно случайная функция, иногда называемая случайным оракулом. В частности, у безопасной хеш-функции не должно быть свойств или закономерностей, которых не было бы у случайной

функции. Это определение полезно теоретикам, но на практике нужны более конкретные понятия: устойчивость к нахождению прообраза и устойчивость к коллизиям.

Устойчивость к нахождению прообраза

Прообразом заданного значения хеша `H` называется любое сообщение `M`, такое что `Hash(M)=H`. Устойчивость к нахождению прообраза означает гарантию безопасности, согласно которой при заданном случайном значении хеша атакующий никогда не найдет его прообраз. Хеш-функции иногда называют односторонними функциями, поскольку от сообщения можно перейти к его хешу, но не наоборот.

В самом деле, обратить хеш-функцию невозможно даже при неограниченной вычислительной мощности. Предположим, я вычислил хеш некоторого сообщения с помощью SHA-256 и получил следующее 256-битное значение:

```
f67a58184cef99d6dfc3045f08645e844f2837ee4bfcc6c949c9f76743667adfd
```

Даже располагая бесконечным временем и вычислительной мощностью, вы не сможете определить, какое сообщение я выбрал для получения именно этого хеша, поскольку одному значению хеша соответствует множество сообщений. Поэтому вы найдете некоторые сообщения, дающие этот хеш, возможно, включая выбранное мной, но не сможете определить, каким именно сообщением воспользовался я. Получается безусловная безопасность.

Например, существует `2^256` возможных значений 256-битного хеша, типичной длины практических хеш-функций, но возможных 1024-битных сообщений намного больше, а именно `2^1024`. Следовательно, каждому возможному 256-битному значению хеша в среднем соответствует

```
frac(2^1024)(2^256)=2^(1024-256)=2^768
```

прообразов длиной 1024 бита.

На практике нужно гарантировать, что найти какое-либо сообщение, соответствующее заданному значению хеша, практически невозможно, а не только невозможно найти то сообщение, которое использовалось изначально. Именно это и означает устойчивость к нахождению прообраза. Более точно различают устойчивость к нахождению первого и второго прообразов. Устойчивость к нахождению первого прообраза, или просто устойчивость к нахождению прообраза, описывает случаи, когда практически невозможно найти сообщение, хеш которого равен заданному значению. Устойчивость к нахождению второго прообраза означает, что для заданного сообщения `M_1` практически невозможно найти другое сообщение `M_2`, хеш которого совпадает с хешем `M_1`.

Стоимость нахождения прообраза

Имея хеш-функцию и значение хеша, первый прообраз можно искать, перебирая разные сообщения, пока одно из них не даст целевой хеш. В листинге 6-1 это делается алгоритмом, похожим на `solve_preimage()`.

```
solve_preimage(H) {
    repeat {
        M = random_message()
        if Hash(M) == H then return M
    }
}
```

Листинг 6-1. Оптимальный алгоритм поиска прообраза для безопасной хеш-функции

Здесь `random_message()` генерирует случайное сообщение, например случайное 1024-битное значение. Если длина хеша `n` достаточно велика, `solve_preimage()` практически никогда не

завершится, поскольку для нахождения прообраза потребуется в среднем 2^n попыток. При n=256, как в современных хешах SHA-256 и BLAKE2, положение безнадежно.

Почему устойчивость ко второму прообразу слабее

Если вы умеете находить первые прообразы, то сможете находить и вторые прообразы для той же хеш-функции. В доказательство предположим, что алгоритм solve_preimage() возвращает прообраз заданного значения хеша. Тогда для нахождения второго прообраза некоторого сообщения M можно использовать алгоритм из листинга 6-2.

```
solve_second_preimage(M) {
    H = Hash(M)
    return solve_preimage(H)
}
```

Листинг 6-2. Нахождение вторых прообразов при наличии способа находить первые прообразы

Второй прообраз находится путем сведения задачи к нахождению прообраза и применения соответствующей атаки. Следовательно, любая хеш-функция, устойчивая к нахождению второго прообраза, устойчива и к нахождению первого. В противном случае она не была бы устойчива и ко второму прообразу, что следует из приведенного выше алгоритма solve_second_preimage(). Иными словами, лучшая атака для нахождения второго прообраза почти идентична лучшей атаке для нахождения первого прообраза, если только в хеш-функции нет дефекта, допускающего более эффективные атаки. Обратите также внимание, что атака по поиску прообраза в основе своей похожа на атаку с восстановлением ключа блочного или потокового шифра, за исключением того, что в случае шифрования существует ровно одно решение известного размера.

Устойчивость к коллизиям

Какую бы хеш-функцию вы ни выбрали, коллизии неизбежно существуют в силу принципа Дирихле: если имеется m клеток и n голубей, которых нужно в них поместить, причем n больше m, то хотя бы в одной клетке окажется больше одного голубя.

> **Примечание.** Принцип Дирихле можно обобщить на другие предметы и вместилища. Например, любая последовательность из 27 слов в Конституции США содержит по меньшей мере два слова, начинающихся с одной и той же буквы. В мире хеш-функций клетки - это значения хеша, а голуби - сообщения. Поскольку возможных сообщений намного больше, чем значений хеша, коллизии неизбежны.

Однако, несмотря на неизбежность коллизий, их должно быть трудно находить, чтобы хеш-функция считалась устойчивой к коллизиям. Иными словами, атакующий не должен быть способен найти два разных сообщения, имеющих одинаковый хеш.

Понятие устойчивости к коллизиям связано с устойчивостью к нахождению второго прообраза: если для хеш-функции можно находить вторые прообразы, то можно находить и коллизии, как показано в листинге 6-3.

```
solve_collision() {
    M = random_message()
    return (M, solve_second_preimage(M))
}
```

Листинг 6-3. Наивный алгоритм поиска коллизии

Иными словами, любая хеш-функция, устойчивая к коллизиям, устойчива и к нахождению второго прообраза. Если бы это было не так, существовал бы эффективный алгоритм нахождения второго

прообраза, с помощью которого можно было бы нарушить устойчивость к коллизиям.

Как находить коллизии

Находить коллизии быстрее, чем прообразы: благодаря атаке дней рождения требуется примерно 2^(n/2) операций вместо 2^n. Ее основная идея заключается в следующем: имея N сообщений и столько же значений хеша, можно получить в общей сложности N*(N-1)/2 потенциальных коллизий, рассматривая каждую пару значений хеша. Это величина того же порядка, что N^2. Атака называется атакой дней рождения, поскольку ее обычно поясняют с помощью парадокса дней рождения: в группе из 23 человек с вероятностью, близкой к 1/2, найдутся два человека с одним днем рождения. Это не парадокс, а лишь неожиданный для многих результат.

> **Примечание.** N*(N-1)/2 - число пар различных сообщений. Деление на 2 нужно потому, что (M_1,M_2) и (M_2,M_1) считаются одной и той же парой: порядок не имеет значения.

Для сравнения, при поиске прообраза N сообщений дают всего N кандидатов на прообраз, тогда как те же N сообщений дают приблизительно N^2 потенциальных коллизий. Поскольку вместо N получается N^2, можно сказать, что шансов найти решение квадратично больше. Соответственно, сложность поиска квадратично ниже: для нахождения коллизии используется квадратный корень из 2^n сообщений, то есть 2^(n/2) вместо 2^n.

Наивная атака дней рождения

Вот простейший способ нахождения коллизий посредством атаки дней рождения:

1. Вычислить 2^(n/2) хешей для 2^(n/2) произвольно выбранных сообщений и сохранить все пары «сообщение/хеш» в списке.
2. Отсортировать список по значениям хеша, чтобы все одинаковые значения оказались рядом.
3. Просмотреть отсортированный список и найти две соседние записи с одинаковым значением хеша.

К сожалению, этот метод требует много памяти - достаточной для хранения 2^(n/2) пар «сообщение/хеш», - а сортировка большого числа элементов замедляет поиск: в среднем требуется около n2^(n/2) операций при использовании быстрого алгоритма сортировки, например быстрой сортировки.

Поиск коллизий с малым расходом памяти методом ро

Метод ро - это алгоритм поиска коллизий, который, в отличие от наивной атаки дней рождения, требует лишь небольшого объема памяти. Он работает следующим образом:

1. Для хеш-функции с n-битными значениями хеша выбирается некоторое случайное значение H_1 и полагается H_1=H'_1.
2. Вычисляются H_2=Hash(H_1) и H'_2=Hash(Hash(H'_1)). В первом случае хеш-функция применяется один раз, а во втором - дважды.
3. Процесс повторяется: для возрастающих значений i вычисляются H_(i+1)=Hash(H_i) и H'_(i+1)=Hash(Hash(H'_i)), пока не будет достигнуто такое i, что H_(i+1)=H'_(i+1).

Атака наглядно показана на рисунке 6-3, где стрелка, например, от H_1 к H_2 означает H_2=Hash(H_1). Обратите внимание, что последовательность значений H_i в конце концов входит в

петлю, также называемую циклом, форма которого напоминает греческую букву ро (rho). Цикл начинается в H_5 и характеризуется коллизией Hash(H_4)=Hash(H_10)=H_5. Главное наблюдение состоит в том, что для нахождения коллизии достаточно найти такой цикл. Метод ро позволяет атакующему определить положение цикла и тем самым найти коллизию.

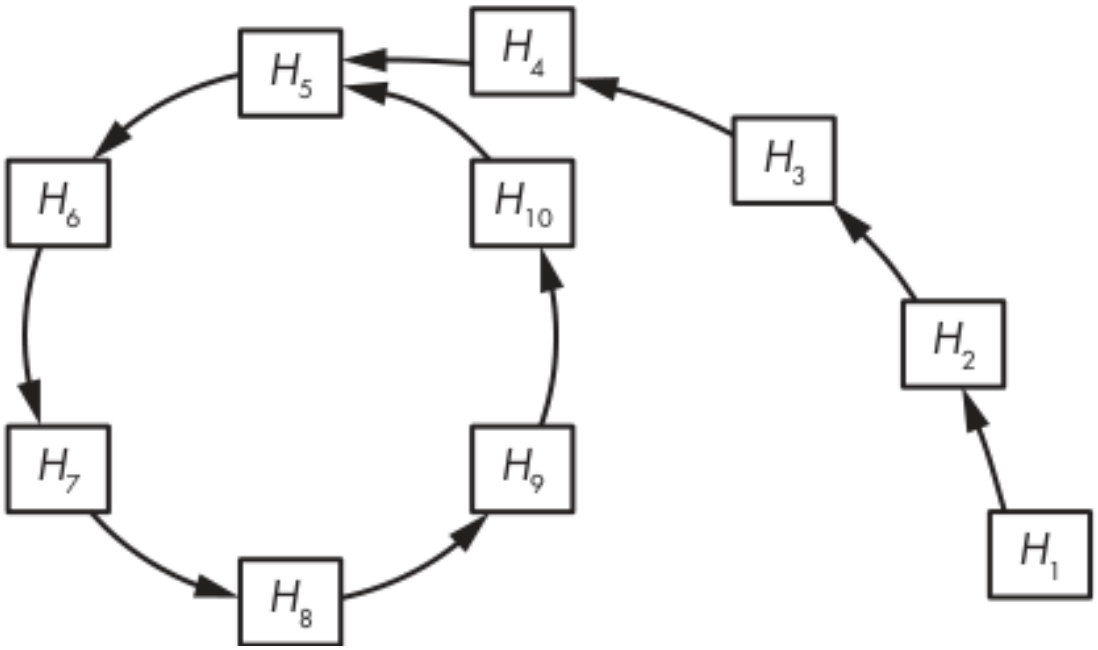

Рисунок 6-3. Структура метода ро для хеш-функции, где каждая стрелка обозначает одно вычисление хеш-функции. Циклу, начинающемуся в H_5, соответствует коллизия Hash(H_4)=Hash(H_10)=H_5

Усовершенствованные методы поиска коллизий, основанные на методе ро, сначала обнаруживают начало цикла, а затем находят коллизию, не сохраняя множество значений в памяти и не сортируя длинный список. Для успеха требуется около 2^(n/2) операций. Разумеется, на рисунке 6-3 показано намного меньше значений хеша, чем у реальной функции с дайджестами длиной 256 битов или более. В среднем и цикл, и хвост, то есть участок от H_1 до H_5 на рисунке 6-3, содержат примерно по 2^(n/2) значений хеша, где n - битовая длина значений. Поэтому для нахождения коллизии потребуется не менее 2^(n/2)+2^(n/2) вычислений хеша.

Как строятся хеш-функции

В 1980-х годах криптографы поняли, что самый простой способ хешировать сообщение - разбить его на фрагменты и последовательно обработать каждый фрагмент похожим алгоритмом. Эта стратегия, называемая итеративным хешированием, имеет две основные формы:

• Итеративное хеширование с функцией сжатия, преобразующей входные данные в более короткий результат, как показано на рисунке 6-4. Этот метод также называется конструкцией Меркла - Дамгора в честь описавших ее криптографов Ральфа Меркла и Ивана Дамгора.
• Итеративное хеширование с функцией, преобразующей входные данные в выходные данные того же размера так, что любые два разных входа дают два разных выхода, то есть с перестановкой. Такие функции называются губчатыми.

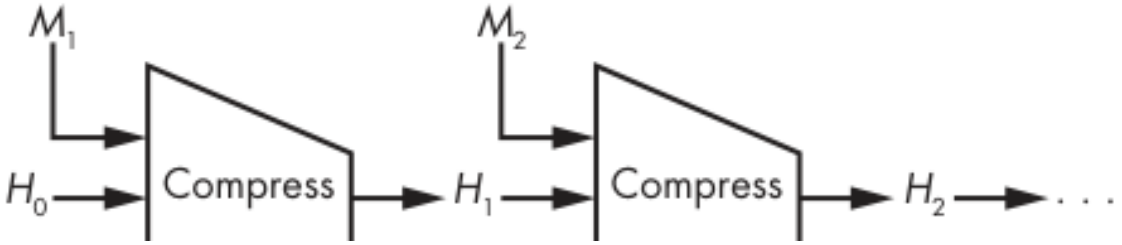

Рисунок 6-4. Конструкция Меркла - Дамгора с функцией сжатия `Compress`

Теперь мы обсудим, как работают эти конструкции и как выглядят функции сжатия на практике.

Хеш-функции на основе сжатия

Все хеш-функции, разработанные с 1980-х по 2010-е годы, основаны на конструкции Меркла - Дамгора (M-D): MD4, MD5, SHA-1 и семейство SHA-2, а также менее известные хеш-функции RIPEMD и Whirlpool. Хотя конструкция M-D несовершенна, она проста и оказалась достаточно безопасной для многих приложений.

> **Примечание.** В названиях MD4, MD5 и RIPEMD буквы MD означают message digest, то есть «дайджест сообщения», а не Merkle - Damgard.

Чтобы хешировать сообщение, конструкция M-D разбивает его на блоки одинакового размера и смешивает эти блоки с внутренним состоянием с помощью функции сжатия, как показано на рисунке 6-4. Здесь `H_0` - начальное значение внутреннего состояния, обозначаемое IV, значения `H_1,H_2,...` - цепочные значения, а итоговое значение внутреннего состояния - значение хеша сообщения.

Размер блоков сообщения часто составляет 512 или 1024 бита, но в принципе может быть любым. Каким бы он ни был, для конкретной хеш-функции он фиксирован. Например, SHA-256 работает с 512-битными блоками, а SHA-512 - с 1024-битными.

Дополнение блоков

Что происходит, если нужно хешировать сообщение, которое нельзя разбить на последовательность полных блоков? Например, если размер блока равен 512 битам, то 520-битное сообщение состоит из одного 512-битного блока и еще 8 битов. В таком случае конструкция M-D формирует последний блок следующим образом: берет оставшийся фрагмент битов, в нашем примере 8 битов, приписывает один единичный бит, затем нулевые биты и, наконец, длину исходного сообщения, закодированную фиксированным числом битов. Этот способ дополнения гарантирует, что любые два разных сообщения дают разные последовательности блоков, а следовательно, разные значения хеша.

Например, если хешировать восьмибитную строку `10101010` с помощью SHA-256, хеш-функции с 512-битными блоками сообщений, первый и единственный блок в битовом представлении будет выглядеть так:

```
10101010100000000000000 ... 0000000000001000
```

Здесь биты сообщения - первые 8 битов (`10101010`), а все последующие биты составляют дополнение. Последовательность `1000` в конце блока - это длина сообщения, то есть число 8 в двоичной записи, кодируемое не более чем 32 битами. Таким образом, дополнение создает

512-битное сообщение, состоящее из единственного 512-битного блока и готовое к обработке функцией сжатия SHA-256.

Гарантии безопасности

Конструкция Меркла - Дамгора превращает безопасную функцию сжатия, принимающую небольшие входные данные фиксированной длины, в безопасную хеш-функцию, принимающую входные данные произвольной длины. Если функция сжатия устойчива к нахождению прообраза и коллизиям, то построенная на ее основе посредством конструкции M-D хеш-функция также устойчива к прообразам и коллизиям. Это верно потому, что любую успешную атаку по нахождению прообраза хеша M-D можно превратить в успешную атаку по нахождению прообраза функции сжатия, как Меркл и Дамгор показали в своих статьях 1989 года, упомянутых в разделе «Дополнительная литература» этой главы. То же верно и для коллизий: атакующий не может нарушить устойчивость хеша к коллизиям, не нарушив устойчивость к коллизиям лежащей в его основе функции сжатия. Следовательно, безопасность последней гарантирует безопасность хеша.

Обратите внимание, что обратное утверждение неверно, поскольку коллизия функции сжатия не обязательно дает коллизию хеша. Произвольная коллизия между `Compress(X,M_1)` и `Compress(Y,M_2)` для цепочных значений `X` и `Y`, отличных от `H_0`, не даст коллизии хеша, поскольку ее нельзя «подключить» к итеративной цепочке хешей. Исключение возможно, если одним из цепочных значений случайно окажется `X`, а другим `Y`, но это маловероятно.

Нахождение мультиколлизий

Мультиколлизия возникает, когда множество из трех или более сообщений хешируется в одно значение. Например, тройка `(X,Y,Z)`, для которой `Hash(X)=Hash(Y)=Hash(Z)`, представляет собой 3-коллизию. В идеале мультиколлизии должны находиться намного труднее, чем коллизии, однако существует простой прием, позволяющий находить их почти с той же стоимостью, что одну коллизию. Он работает следующим образом:

1. Найти коллизию `Compress(H_0,M_(1.1))=Compress(H_0,M_(1.2))=H_1`. Это обычная 2-коллизия, то есть два сообщения, хешируемые в одно значение.
2. Найти вторую коллизию с начальным цепочным значением `H_1`: `Compress(H_1,M_(2.1))=Compress(H_1,M_(2.2))=H_2`. Теперь имеется 4-коллизия: четыре сообщения, хешируемые в одно значение `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. Повторить поиск коллизии `N` раз. В результате получится `2^N` сообщений по `N` блоков, хешируемых в одно значение, то есть `2^N`-коллизия, и это обойдется «всего» приблизительно в `N2^(n/2)` вычислений хеша.

На практике этот прием не слишком практичен, поскольку сначала все равно требуется найти базовую 2-коллизию.

Построение функций сжатия

Все функции сжатия, применяемые в реальных хеш-функциях, таких как SHA-256 и BLAKE2, основаны на блочных шифрах, потому что это самый простой способ построения функции сжатия. На рисунке 6-5 показана самая распространенная функция сжатия на основе блочного шифра - конструкция Дэвиса - Мейера.

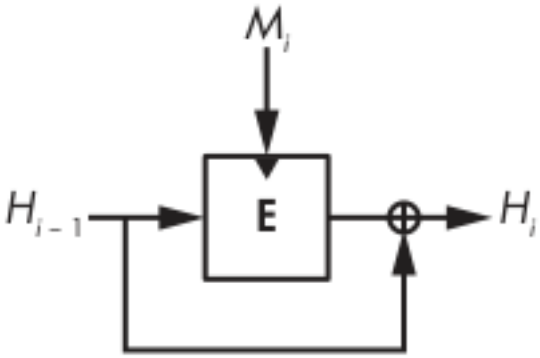

Рисунок 6-5. Конструкция Дэвиса - Мейера. Темный треугольник показывает вход ключа блочного шифра

Имея блок сообщения `M_i` и предыдущее цепочное значение `H_(i-1)`, функция сжатия Дэвиса - Мейера использует блочный шифр `E` для вычисления нового цепочного значения:

```
H_i=E(M_i,H_(i-1)) xor H_(i-1).
```

Блок сообщения `M_i` играет роль ключа блочного шифра, а цепочное значение `H_(i-1)` - блока открытого текста. Пока блочный шифр безопасен, получившаяся функция сжатия также безопасна и устойчива к коллизиям и нахождению прообраза. Без XOR с предшествующим цепочным значением `xor H_(i-1)` конструкция Дэвиса - Мейера была бы небезопасной, поскольку ее можно было бы обратить: перейти от нового цепочного значения к предыдущему с помощью функции расшифрования блочного шифра.

> **Примечание.** Конструкция Дэвиса - Мейера обладает удивительным свойством: для нее можно находить неподвижные точки, то есть цепочные значения, которые не меняются после применения функции сжатия с заданным блоком сообщения. Достаточно взять в качестве цепочного значения `H_(i-1)=D(M_i,0)`, где `D` - функция расшифрования, соответствующая `E`. Тогда новое цепочное значение `H_i` равно исходному `H_(i-1)`:

```
H_i =E(M_i,H_(i-1)) xor H_(i-1)
=E(M_i,D(M_i,0)) xor D(M_i,0)
=0 xor D(M_i,0)=D(M_i,0)=H_(i-1).
```

> Равенство `H_i=H_(i-1)` получается потому, что подстановка расшифрованного нуля в функцию шифрования дает ноль: член `E(M_i,D(M_i,0))=0`, и в выражении результата функции сжатия остается лишь часть `xor H_(i-1)`. Таким образом, можно находить неподвижные точки, например, для функций сжатия семейства SHA-2, основанных на конструкции Дэвиса - Мейера. К счастью, неподвижные точки не создают угрозы безопасности.

Помимо конструкции Дэвиса - Мейера существует множество других функций сжатия на основе блочного шифра, в том числе показанные на рисунке 6-6.

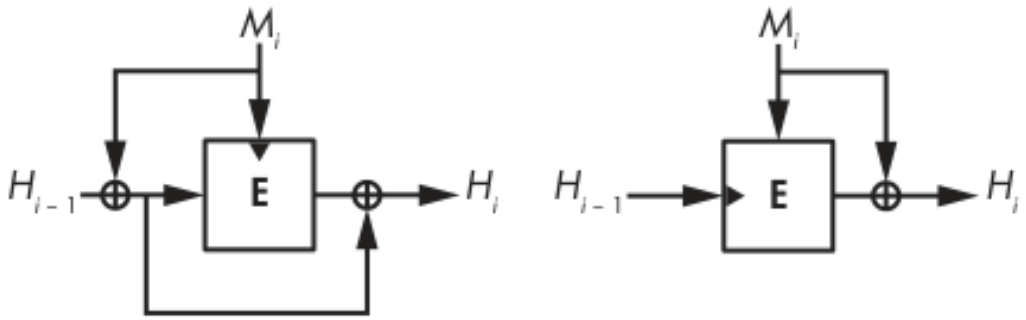

Рисунок 6-6. Другие безопасные конструкции функций сжатия на основе блочного шифра

Они менее популярны, поскольку сложнее или требуют, чтобы блок сообщения имел ту же длину, что цепочное значение.

Хеш-функции на основе перестановки

После десятилетий исследований криптографы знают о методах хеширования на основе блочных шифров практически все. Но разве нельзя хешировать проще? Зачем использовать блочный шифр - алгоритм, принимающий секретный ключ, - если хеш-функции секретного ключа не принимают? Почему бы не строить хеш-функции на основе блочного шифра с фиксированным ключом, то есть единственного алгоритма перестановки?

Такими более простыми хеш-функциями являются губчатые функции. Вместо функции сжатия и блочного шифра они используют одну перестановку, как показано на рисунке 6-7. Вместо блочного шифра для смешивания битов сообщения с внутренним состоянием губчатые функции выполняют обычную операцию XOR. Губчатые функции не только проще функций Меркла - Дамгора, но и универсальнее. Они применяются как хеш-функции, а также как генераторы детерминированных случайных битов, потоковые шифры, псевдослучайные функции, рассматриваемые в главе 7, и аутентифицированные шифры, рассматриваемые в главе 8. Самая известная губчатая функция - Keccak, также называемая SHA-3.

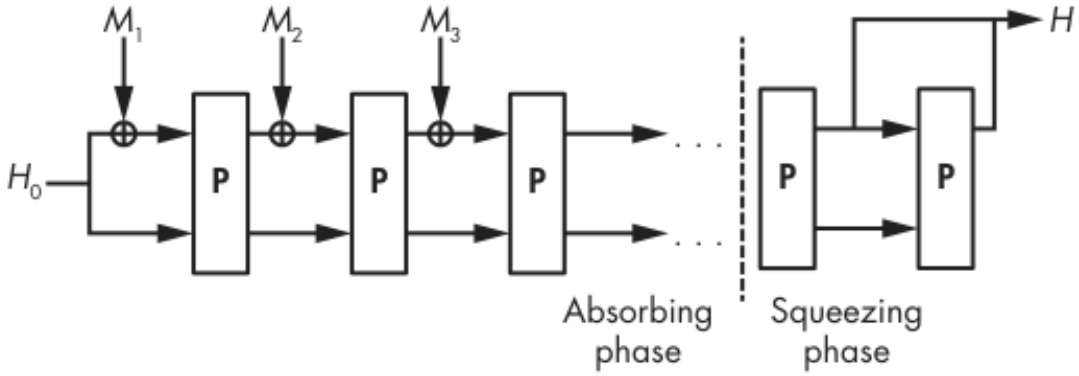

Рисунок 6-7. Губчатая конструкция

Губчатая функция работает следующим образом:

1. Первый блок сообщения `M_1` складывается посредством XOR с `H_0` - заранее определенным начальным значением внутреннего состояния, например строкой из одних нулей. Все блоки сообщения имеют один размер, меньший размера внутреннего состояния.
2. Перестановка `P` преобразует внутреннее состояние в другое значение того же размера.
3. Блок `M_2` складывается посредством XOR с состоянием, применяется `P`, после чего действия повторяются для блоков `M_3`, `M_4` и так далее. Это фаза поглощения.

4. После ввода всех блоков сообщения снова применяется P, а затем из состояния извлекается блок битов, образующий хеш. Если нужен более длинный хеш, P применяется еще раз и извлекается следующий блок. Это фаза выжимания.

Безопасность губчатой функции зависит от длины ее внутреннего состояния и размера блоков. Если длина блоков сообщения равна r битам, а длина внутреннего состояния - w битам, то c=w-r битов внутреннего состояния недоступны для изменения блоками сообщения. Значение c называется емкостью губки, а гарантируемый губчатой функцией уровень безопасности равен c/2. Например, для достижения 256-битной безопасности при 64-битных блоках сообщения внутреннее состояние должно иметь длину не менее w=2*256+64=576 битов. Уровень безопасности зависит также от длины n значения хеша. Поэтому сложность атаки по поиску коллизии равна меньшему из значений 2^(n/2) и 2^(c/2), а сложность атаки по нахождению второго прообраза - меньшему из 2^n и 2^(c/2).

Для безопасности перестановка P должна вести себя как случайная: не иметь статистических смещений и математической структуры, позволяющей атакующему предсказывать выходные данные. Подобно хешам на основе функций сжатия, губчатые функции тоже дополняют сообщения, но дополнение проще, поскольку в него не нужно включать длину сообщения. За последним битом сообщения просто приписывается единичный бит, а затем столько нулей, сколько необходимо.

Семейство хеш-функций SHA

Хеш-функции Secure Hash Algorithm (SHA) - это стандарты, определенные NIST для применения невоенными федеральными ведомствами США. Они считаются мировыми стандартами, и лишь некоторые правительства за пределами США выбирают собственные алгоритмы хеширования, например китайский SM3, российский «Стрибог» и украинскую «Купину», скорее из соображений суверенитета, чем из-за недоверия к безопасности SHA. Американские SHA исследованы криптоаналитиками намного тщательнее, чем их зарубежные аналоги.

> **Примечание.** Message Digest 5 (MD5) была самой популярной хеш-функцией с 1992 года до ее взлома примерно в 2005 году, после чего многие приложения перешли на одну из хеш-функций SHA. MD5 обрабатывает 512-битные блоки сообщений и обновляет 128-битное внутреннее состояние, формируя 128-битный хеш. Таким образом, она обеспечивает в лучшем случае 128-битную устойчивость к нахождению прообраза и 64-битную устойчивость к коллизиям. В 1996 году криптоаналитики предупредили о коллизии функции сжатия MD5, но предупреждение оставалось без внимания до 2005 года, когда группа китайских криптоаналитиков научилась вычислять коллизии полной хеш-функции MD5. На момент написания книги коллизию MD5 можно найти всего за несколько секунд, однако некоторые системы по-прежнему используют или поддерживают MD5, часто ради обратной совместимости.

SHA-1

Стандарт SHA-1 появился вследствие неудачи исходной хеш-функции SHA-0, разработанной АНБ. В 1993 году NIST стандартизировал алгоритм SHA-0, но в 1995 году АНБ выпустило SHA-1, исправив неуказанную проблему безопасности SHA-0. Причина изменения прояснилась в 1998 году, когда двое исследователей нашли способ поиска коллизий SHA-0 примерно за `2^60` операций вместо `2^80`, ожидаемых для 160-битных хеш-функций вроде SHA-0 и SHA-1. Более поздние атаки снизили сложность примерно до `2^33` операций, что позволило получать реальные коллизии SHA-0 менее чем за час.

Внутреннее устройство SHA-1

SHA-1 объединяет хеш-функцию Меркла - Дамгора с функцией сжатия Дэвиса - Мейера, основанной на специально созданном блочном шифре. Иными словами, SHA-1 последовательно выполняет над 512-битными блоками сообщения `M` следующую операцию:

```
H=E(M,H)+H.
```

Знак сложения +, а не `xor` (XOR), использован здесь намеренно. `E(M,H)` и `H` рассматриваются как массивы 32-битных целых чисел, и слова в одинаковых позициях складываются друг с другом: первое 32-битное слово `E(M,H)` - с первым 32-битным словом `H` и так далее. Начальное значение `H` постоянно для любого сообщения, затем `H` изменяется в соответствии с приведенным уравнением, а итоговое значение `H` после обработки всех блоков возвращается как хеш сообщения.

После выполнения блочного шифра с блоком сообщения в роли ключа и текущим 160-битным цепочным значением в роли блока открытого текста 160-битный результат рассматривается как массив из пяти 32-битных слов. Каждое из них складывается с соответствующим 32-битным словом исходного значения `H`.

В листинге 6-4 показана функция сжатия SHA-1 `SHA1-compress()`:

```
SHA1-compress(H, M) {
    (a0, b0, c0, d0, e0) = H   // Разобрать H как пять 32-битных слов с порядком от
    старшего байта
    (a, b, c, d, e) = SHA1-blockcipher(a0, b0, c0, d0, e0, M)
    return (a + a0, b + b0, c + c0, d + d0, e + e0)
}
```

Листинг 6-4. Функция сжатия SHA-1

Блочный шифр SHA-1 `SHA1-blockcipher()` принимает 512-битный блок сообщения `M` как ключ и преобразует пять 32-битных слов `a`, `b`, `c`, `d` и `e`, 80 раз повторяя короткую последовательность операций, чтобы заменить слово `a` комбинацией всех пяти слов. Затем остальные слова массива сдвигаются, как в регистре сдвига. В листинге 6-5 эти операции описаны псевдокодом, где `K[i]` - зависящая от раунда константа.

```
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)
}
```

Листинг 6-5. Блочный шифр SHA-1

Функция `expand()` из листинга 6-5 создает массив `W` из 80 32-битных слов на основе 16-словного блока сообщения: первые 16 слов `W` берутся из `M`, а последующие вычисляются посредством XOR предыдущих слов с циклическим сдвигом на один бит влево. Соответствующий псевдокод показан

в листинге 6-6.

```
expand(M) {
    // 512-битное M рассматривается как массив из шестнадцати 32-битных слов.
    W = empty array of eighty 32-bit words
    for i = 0 to 79 {
        if i < 16 then W[i] = M[i]
        else
            W[i] = (W[i - 3] XOR W[i - 8] XOR W[i - 14] XOR W[i - 16]) <<< 1
    }
    return W
}
```

Листинг 6-6. Функция `expand()` SHA-1

Операция `<<< 1` в листинге 6-6 - единственное различие между функциями SHA-1 и SHA-0.

Наконец, в листинге 6-7 показана функция `f()` из `SHA1-blockcipher()` - последовательность базовых побитовых логических операций, то есть булева функция, зависящая от номера раунда.

```
f(i, b, c, d) {
    if i < 20 then return ((b & c) XOR (~b & d))
    if i < 40 then return (b XOR c XOR d)
    if i < 60 then return ((b & c) XOR (b & d) XOR (c & d))
    if i < 80 then return (b XOR c XOR d)
}
```

Листинг 6-7. Функция `f()` SHA-1

Вторая и четвертая булевы функции в листинге 6-7 просто складывают три входных слова посредством XOR, а это линейная операция. В отличие от них первая и третья функции используют нелинейный оператор &, то есть логическое AND, для защиты от дифференциального криптоанализа, эксплуатирующего предсказуемое распространение побитовой разности. Без оператора &, например если бы `f()` всегда равнялась `b xor c xor d`, SHA-1 было бы легко взломать, прослеживая закономерности во внутреннем состоянии.

Атаки на SHA-1

Хотя SHA-1 надежнее SHA-0, она также небезопасна. Поэтому с 2014 года браузер Chrome помечает сайты, использующие SHA-1 в HTTPS-соединениях, как небезопасные. Хотя 160-битный хеш должен обеспечивать 80-битную устойчивость к коллизиям, в 2005 году исследователи обнаружили слабости SHA-1 и оценили сложность поиска коллизии приблизительно в `2^63` вычислений против `2^80` для безупречного алгоритма. Реальная коллизия SHA-1 появилась лишь через 12 лет, когда после многолетних исследований криптоаналитики совместно с исследователями Google представили два коллидирующих PDF-документа; см. https://shattered.io.

Не следует использовать SHA-1. Большинство веб-браузеров теперь помечают SHA-1 как небезопасную, и NIST больше не рекомендует ее. Вместо нее используйте SHA-2, SHA-3, BLAKE2 или BLAKE3.

SHA-2

SHA-2, преемница SHA-1, была разработана АНБ и стандартизирована NIST в 2002 году. SHA-2 - семейство из четырех хеш-функций: SHA-224, SHA-256, SHA-384 и SHA-512, причем SHA-256 и SHA-512 являются двумя основными алгоритмами. Трехзначные числа обозначают битовую длину каждого хеша.

Первоначальная цель разработки SHA-2 состояла в создании более длинных хешей, обеспечивающих более высокие уровни безопасности, чем SHA-1. Однако конструкции алгоритмов

SHA-1 и SHA-2 похожи. Все варианты SHA-2 также используют конструкцию Меркла - Дамгора и функцию сжатия, близкую к функции SHA-1, но с более сильной нелинейностью и лучшим распространением различий.

SHA-256

SHA-256 - самый распространенный вариант SHA-2. У SHA-1 цепочные значения имеют длину 160 битов, а у SHA-256 - 256 битов и представлены восемью 32-битными словами. И SHA-1, и SHA-256 имеют 512-битные блоки сообщений, но SHA-1 выполняет 80 раундов, тогда как SHA-256 - 64, расширяя 16-словный блок сообщения до 64-словного с помощью функции `expand256()`, показанной в листинге 6-8.

```
expand256(M) {
    // 512-битное M рассматривается как массив из шестнадцати 32-битных слов.
    W = empty array of sixty-four 32-bit words
    for i = 0 to 63 {
        if i < 16 then W[i] = M[i]
        else {
            // Операция ">>" является сдвигом, а не циклическим сдвигом ">>>"; это не
    опечатка.
            s0 = (W[i - 15] >>> 7) XOR (W[i - 15] >>> 18) XOR (W[i - 15] >> 3)
            s1 = (W[i - 2] >>> 17) XOR (W[i - 2] >>> 19) XOR (W[i - 2] >> 10)
            W[i] = W[i - 16] + s0 + W[i - 7] + s1
        }
    }
    return W
}
```

Листинг 6-8. Функция `expand256()` SHA-256

Обратите внимание, насколько расширение сообщения `expand256()` в SHA-2 сложнее, чем `expand()` SHA-1 из листинга 6-6, где выполняются лишь операции XOR и циклический сдвиг на один бит. Главный цикл функции сжатия SHA-256 также сложнее цикла SHA-1: за одну итерацию он выполняет 26 арифметических операций против 11 у SHA-1. Это снова XOR, логические AND и циклические сдвиги слов. Повышенная сложность делает SHA-256 устойчивее к дифференциальному криптоанализу.

Другие алгоритмы SHA-2

Семейство SHA-2 включает SHA-224, которая алгоритмически идентична SHA-256, за исключением того, что ее начальное значение представляет собой другой набор из восьми 32-битных слов, а длина значения хеша равна 224, а не 256 битам: берутся первые 224 бита итогового цепочного значения.

В семейство SHA-2 входят также алгоритмы SHA-512 и SHA-384. SHA-512 похожа на SHA-256, но работает с 64-битными словами вместо 32-битных. Поэтому в ней используются 512-битные цепочные значения, то есть восемь 64-битных слов, принимаются 1024-битные блоки сообщений, то есть шестнадцать 64-битных слов, и выполняется 80 раундов вместо 64. В остальном функция сжатия почти такая же, как у SHA-256, только расстояния циклических сдвигов изменены с учетом большей ширины слова. Например, SHA-512 включает операцию `a >>> 34`, которая не имела бы смысла для 32-битных слов SHA-256. SHA-384 относится к SHA-512 так же, как SHA-224 к SHA-256: это тот же алгоритм, но с другим начальным значением и итоговым хешем, усеченным до 384 битов.

С точки зрения безопасности все четыре версии SHA-2 пока оправдывают ожидания: SHA-256 гарантирует 256-битную устойчивость к нахождению прообраза, SHA-512 - приблизительно

256-битную устойчивость к коллизиям и так далее. Однако настоящего доказательства безопасности функций SHA-2 не существует: речь идет о предполагаемой безопасности.

Тем не менее после практических атак на MD5 и SHA-1 исследователи и NIST обеспокоились долговременной безопасностью SHA-2 из-за ее сходства с SHA-1, и многие полагали, что атаки на SHA-2 - лишь вопрос времени. На момент написания книги успешной атаки на SHA-2 еще не появилось. Как бы то ни было, NIST разработал запасной план: SHA-3.

Конкурс SHA-3

Объявленный в 2007 году конкурс хеш-функций NIST, официальное название конкурса SHA-3, начался с приема заявок и нескольких базовых требований: предложенные хеш-функции должны были быть не менее безопасными и быстрыми, чем SHA-2, и уметь как минимум все то же, что SHA-2. Кандидаты SHA-3 также не должны были слишком походить на SHA-1 и SHA-2, чтобы оставаться невосприимчивыми к атакам, способным взломать SHA-1 и потенциально SHA-2. К 2008 году NIST получил 64 предложения со всего мира, в том числе от университетов и крупных корпораций: BT, IBM, Microsoft, Qualcomm, Sony и других. Из этих 64 предложений 51 соответствовало требованиям и прошло в первый раунд конкурса.

В первые недели конкурса криптоаналитики безжалостно атаковали предложенные конструкции. В июле 2009 года NIST объявил 14 участников второго раунда. После 15 месяцев анализа и оценки производительности кандидатов NIST выбрал пять финалистов:

BLAKE. Улучшенный хеш Меркла - Дамгора, функция сжатия которого основана на блочном шифре, в свою очередь основанном на основной функции потокового шифра ChaCha - цепочке сложений, XOR и циклических сдвигов слов. BLAKE разработала группа исследователей из университетов Швейцарии и Великобритании, включая меня в то время, когда я был аспирантом.

Grøstl. Улучшенный хеш Меркла - Дамгора, функция сжатия которого использует две перестановки, или блочные шифры с фиксированным ключом, основанные на блочном шифре AES. Grøstl разработала группа из семи исследователей из университетов Дании и Австрии.

JH. Модифицированная губчатая конструкция, в которой блоки сообщения вводятся до и после перестановки, а не только до нее. Перестановка также выполняет операции, похожие на блочный шифр подстановки - перестановки, рассмотренный в главе 4. JH разработал криптограф из университета Сингапура.

Keccak. Губчатая функция, перестановка которой выполняет только побитовые операции. Keccak разработала группа из четырех криптографов, работавших в полупроводниковой компании в Бельгии и Италии; в нее входил один из двух разработчиков AES.

Skein. Хеш-функция, основанная на режиме работы, отличном от Меркла - Дамгора, с функцией сжатия на основе нового блочного шифра, использующего только целочисленное сложение, XOR и циклический сдвиг слов. Skein разработала группа из восьми криптографов из научной и промышленной среды, все, кроме одного, из США; среди них был известный Брюс Шнайер.

После всестороннего анализа пяти финалистов NIST объявил победителя: Keccak. В отчете NIST Keccak был отмечен за «элегантную конструкцию, большой запас безопасности, хорошую общую производительность, превосходную эффективность аппаратной реализации и гибкость». Посмотрим, как работает Keccak.

Keccak (SHA-3)

Одна из причин, по которым NIST выбрал Keccak, состоит в его полном отличии от SHA-1 и SHA-2. Во-первых, это губчатая функция. Основной алгоритм Keccak представляет собой перестановку 1600-битного состояния, принимающую блоки длиной 1152, 1088, 832 или 576 битов и формирующую значения хеша соответственно длиной 224, 256, 384 или 512 битов - тех же четырех размеров, которые дают хеш-функции SHA-2. Но, в отличие от SHA-2, SHA-3 использует один основной алгоритм для всех четырех размеров хеша, а не два.

Другая причина заключается в том, что Keccak - это не просто хеш. Стандарт SHA-3 FIPS 202 определяет четыре хеша - SHA3-224, SHA3-256, SHA3-384 и SHA3-512 - и два алгоритма SHAKE128 и SHAKE256. Название SHAKE означает Secure Hash Algorithm with Keccak. Эти два алгоритма являются функциями с расширяемым выходом (extendable-output functions, XOF), то есть хеш-функциями, способными выдавать хеши переменной, в том числе очень большой длины. Числа 128 и 256 обозначают уровень безопасности каждого алгоритма.

Сам стандарт FIPS 202 объемен и труден для восприятия, однако существуют достаточно быстрые реализации с открытым исходным кодом, позволяющие легко понять алгоритм. Например, tiny_sha3 Маркку-Юхани О. Сааринена по адресу https://github.com/mjosaarinen/tiny_sha3 выражает основной алгоритм Keccak 19 строками кода на C, часть которых воспроизведена в листинге 6-9.

```
static void sha3_keccakf(uint64_t st[25], int rounds)
{
    for (r = 0; r < rounds; r++) {

        // 1 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;
        }

        // 2 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];
        }

        // 3 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];
        }

        // 4 Iota
        st[0] ^= keccakf_rndc[r];
    }
}
```

Листинг 6-9. Реализация tiny_sha3

Программа tiny_sha3 реализует перестановку `P` алгоритма Keccak - обратимое преобразование 1600-битного состояния, рассматриваемого как массив из двадцати пяти 64-битных слов. Код выполняет последовательность раундов, каждый из которых состоит из четырех основных этапов:

1. **Theta** включает операции XOR между 64-битными словами или значениями этих слов, циклически сдвинутыми на один бит; операция `ROTL64(w, 1)` циклически сдвигает слово `w` на один бит влево.
2. **Rho Pi** включает циклические сдвиги 64-битных слов на константы, жестко заданные в массиве `keccakf_rotc[]`.
3. **Chi** включает дополнительные XOR, а также логические AND, оператор &, между 64-битными словами. Эти AND - единственные нелинейные операции в Keccak, и именно они обеспечивают криптографическую стойкость.
4. **Iota** включает XOR с 64-битной константой, жестко заданной в `keccakf_rndc[]`.

Эти операции обеспечивают SHA-3 сильный алгоритм перестановки без смещений и пригодной для эксплуатации структуры. SHA-3 - результат более чем десятилетия исследований, и сотни квалифицированных криптоаналитиков не смогли ее взломать. Маловероятно, что это произойдет в ближайшее время.

Хеш-функции BLAKE2 и BLAKE3

Безопасность важнее всего, но на втором месте стоит скорость. Я видел множество случаев, когда разработчик не переходил с MD5 на SHA-1 просто потому, что MD5 быстрее, или с SHA-1 на SHA-2, потому что SHA-2 заметно медленнее SHA-1. К сожалению, SHA-3 не быстрее SHA-2, а поскольку SHA-2 все еще безопасна, стимулов переходить на SHA-3 немного. Как же хешировать быстрее SHA-1 и SHA-2 и при этом еще безопаснее? Ответ - хеш-функция BLAKE2, выпущенная после конкурса SHA-3.

> **Примечание.** Для полной ясности: я являюсь одним из разработчиков BLAKE2 вместе с Самуэлем Невесом, Зуко Уилкоксом-О'Хирном и Кристианом Виннерлайном.

При разработке BLAKE2 исходили из следующих идей:

• Она должна быть не менее безопасной, чем SHA-3, а возможно, и безопаснее.
• Она должна быть быстрее всех предыдущих стандартов хеширования, включая MD5.
• Она должна подходить для современных приложений и уметь хешировать большие объемы данных как в виде нескольких крупных сообщений, так и множества мелких, с секретным ключом или без него.
• Она должна подходить для современных CPU, поддерживающих параллельные вычисления как на многоядерных системах, так и на уровне инструкций внутри одного ядра.

Результатом инженерной работы стала пара основных хеш-функций:

• BLAKE2b, или просто BLAKE2, оптимизирована для 64-битных платформ и формирует дайджесты длиной от 1 до 64 байтов.
• BLAKE2s оптимизирована для платформ с разрядностью от 8 до 32 битов и формирует дайджесты длиной от 1 до 32 байтов.

У каждой функции есть параллельный вариант, способный использовать несколько ядер CPU. Параллельный аналог BLAKE2b, BLAKE2bp, работает на четырех ядрах, а BLAKE2sp - на восьми. Первая быстрее всего работает на современных CPU серверов и ноутбуков и способна хешировать на CPU ноутбука со скоростью, близкой к 2 Гбит/с. Скорость и возможности сделали BLAKE2 самым популярным хешем, не являющимся стандартом NIST. BLAKE2 используется в бесчисленном

множестве программных приложений и интегрирована в основные криптографические библиотеки, такие как OpenSSL и Sodium. BLAKE2 также является стандартом Инженерного совета Интернета (IETF), описанным в RFC 7693.

> **Примечание.** Спецификации и эталонный код BLAKE2 находятся на https://blake2.net, а оптимизированный код и библиотеки можно загрузить с https://github.com/BLAKE2. Эталонный код также содержит BLAKE2X - расширение BLAKE2, способное формировать значения хеша произвольной длины.

Как показано на рисунке 6-8, функция сжатия BLAKE2 представляет собой вариант конструкции Дэвиса - Мейера, принимающий дополнительные параметры: счетчик, благодаря которому каждое вычисление функции сжатия ведет себя как отдельная функция, и флаг, указывающий, обрабатывает ли функция сжатия последний блок сообщения, что повышает безопасность.

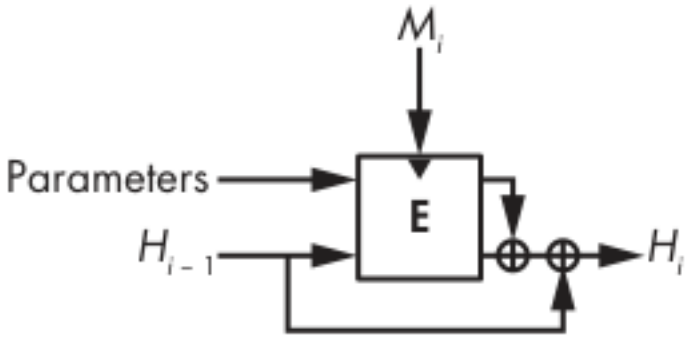

Рисунок 6-8. Функция сжатия BLAKE2. Две половины состояния складываются посредством XOR после блочного шифра

Блочный шифр в функции сжатия BLAKE2 основан на потоковом шифре ChaCha, который сам является вариантом потокового шифра Salsa20, рассмотренного в главе 5. Внутри этого блочного шифра основная операция BLAKE2b состоит из следующей цепочки действий, преобразующей состояние из четырех 64-битных слов с помощью двух слов сообщения `M_i` и `M_j`:

```
a =a+b+M_i,
d =(d xor a) >>> 32,
c =c+d,
b =(b xor c) >>> 24,
a =a+b+M_j,
d =(d xor a) >>> 16,
c =c+d,
b =(b xor c) >>> 63.
```

Основная операция BLAKE2s похожа, но работает с 32-битными, а не 64-битными словами и потому использует другие величины циклических сдвигов.

И последнее, но не менее важное: BLAKE3 - лучше распараллеливаемая, более простая, универсальная и быстрая версия BLAKE2, представленная в 2020 году на конференции Real World Crypto. BLAKE3, разработанная Джеком О'Коннором, Самуэлем Невесом, Зуко Уилкоксом-О'Хирном и мной, быстро стала одной из самых популярных хеш-функций благодаря своим неоспоримым преимуществам. Подробности см. на https://github.com/BLAKE3-team/BLAKE3.

Что может пойти не так

Несмотря на кажущуюся простоту, хеш-функции могут вызвать серьезные проблемы безопасности при использовании не в том месте или не тем способом. Пример - применение слабых алгоритмов контрольных сумм вроде CRC вместо криптографического хеша для проверки целостности файлов в приложениях, передающих данные по сети. Однако эта слабость меркнет по сравнению с другими, способными полностью скомпрометировать кажущиеся безопасными хеш-функции. Вы увидите два примера сбоев: первый относится к SHA-1 и SHA-2, но не к BLAKE2 или SHA-3, а второй - ко всем четырем функциям.

Атака с удлинением сообщения

На рисунке 6-9 показана атака с удлинением сообщения - главная угроза конструкции Меркла - Дамгора.

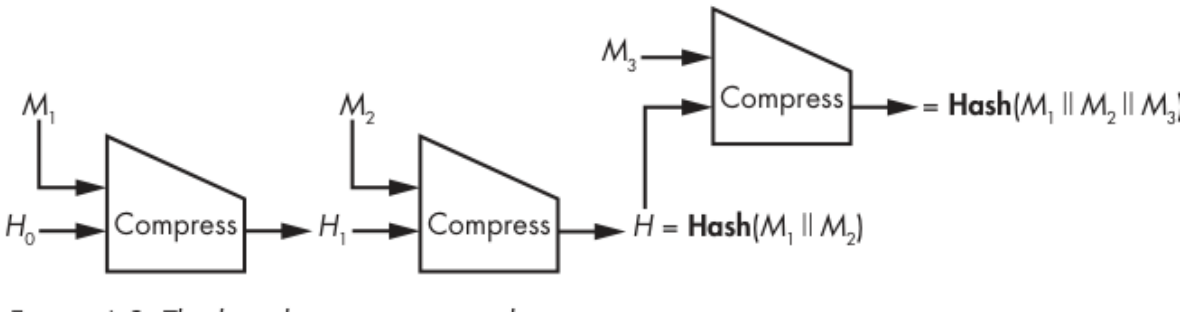

Рисунок 6-9. Атака с удлинением сообщения

В сущности, если известен `Hash(M)` некоторого неизвестного сообщения `M`, состоящего после дополнения из блоков `M_1` и `M_2`, можно определить `Hash(M_1 || M_2 || M_3)` для любого блока `M_3`. Поскольку хеш `M_1 || M_2` является цепочным значением, непосредственно следующим за `M_2`, к хешируемому сообщению можно добавить еще один блок `M_3`, даже не зная хешированных данных. Более того, этот прием обобщается на любое число блоков как в неизвестном сообщении, здесь `M_1 || M_2`, так и в суффиксе, здесь `M_3`.

Атака с удлинением сообщения не затрагивает большинство применений хеш-функций, но способна нарушить безопасность, если использовать хеш слишком изобретательно. К сожалению, хеш-функции SHA-2 уязвимы к этой атаке, хотя АНБ разработало функции, а NIST стандартизировал их, прекрасно зная об изъяне. Его можно было бы устранить, просто сделав последний вызов функции сжатия отличным от всех остальных, например передавая ему дополнительный единичный бит, тогда как предыдущим вызовам передается нулевой. Именно так поступает BLAKE2.

Обман протоколов доказательства хранения

Приложения облачных вычислений используют хеш-функции в протоколах доказательства хранения, то есть протоколах, в которых сервер, поставщик облачных услуг, доказывает клиенту, пользователю облачного хранилища, что действительно хранит файлы, которые обязан хранить от имени клиента.

В 2007 году в статье Рамакришны Котлы, Лоренцо Алвизи и Майка Далина «SafeStore: A Durable and Practical Storage System» (PDF) был предложен следующий протокол доказательства хранения для проверки хранения некоторого файла `M`:

1. Клиент выбирает случайное значение `C` в качестве вызова.
2. Сервер в качестве ответа вычисляет `Hash(M || C)` и отправляет результат клиенту.
3. Клиент также вычисляет `Hash(M || C)` и проверяет совпадение со значением, полученным от сервера.

Исходное предположение статьи состоит в том, что сервер не сможет обмануть клиента: не зная `M`, он не угадает `Hash(M || C)`. Но есть одна тонкость: в действительности `Hash` - итеративный хеш, обрабатывающий входные данные блок за блоком и вычисляющий между блоками промежуточные цепочные значения. Например, если `Hash` - SHA-256, а длина `M` равна 512 битам, размеру блока SHA-256, сервер может обмануть клиента. Как? В первый раз, получив `M`, сервер вычисляет `H_1=Compress(H_0,M_1)` - цепочное значение, полученное из начального значения SHA-256 `H_0` и 512-битного `M`. Затем он сохраняет `H_1` в памяти и удаляет `M`, после чего больше не хранит `M`.

Когда клиент отправляет случайное значение `C`, сервер вычисляет `Compress(H_1,C)`, предварительно дополнив `C` до полного блока, и возвращает результат как `Hash(M || C)`. Клиент считает, что, раз сервер вернул правильное значение `Hash(M || C)`, он хранит полное сообщение, хотя, как вы увидели, это может быть не так.

Этот прием работает для SHA-1, SHA-2, а также SHA-3 и BLAKE2. Решение просто: запрашивать `Hash(C || M)` вместо `Hash(M || C)`.

Дополнительная литература

Чтобы больше узнать о хеш-функциях, прочитайте классические работы 1980-х и 1990-х годов: исследовательские статьи Ральфа Меркла «One Way Hash Functions and DES» и Ивана Дамгора «A Design Principle for Hash Functions». Прочитайте также первое всестороннее исследование хеширования на основе блочных шифров - «Hash Functions Based on Block Ciphers: A Synthetic Approach» Барта Пренела, Рене Говартса и Йоса Вандевалле.

Подробнее о поиске коллизий см. в статье Пола ван Орсхота и Майкла Винера 1997 года «Parallel Collision Search with Cryptanalytic Applications». Чтобы больше узнать о теоретических понятиях безопасности, лежащих в основе устойчивости к нахождению прообраза и коллизиям, а также об атаках с удлинением сообщения, ищите материалы по неразличимости.

Более свежие исследования хеш-функций находятся в архивах конкурса SHA-3, включающих все предложенные алгоритмы и сведения о том, как они были взломаны. Множество ссылок собрано в SHA-3 Zoo по адресу https://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.html и на странице NIST https://csrc.nist.gov/projects/hash-functions/sha-3-project.

Подробнее о победителе конкурса SHA-3 Keccak и губчатых функциях см. на официальной странице разработчиков Keccak https://keccak.team/sponge_duplex.html.

Наконец, можно изучить два реальных случая эксплуатации слабых хеш-функций:

• Вредоносная программа государственного уровня Flame использовала коллизию MD5, чтобы создать поддельный сертификат и выглядеть легитимной программой.
• В игровой консоли Xbox для построения хеш-функции применялся слабый блочный шифр TEA; этот недостаток использовали, чтобы взломать консоль и выполнять на ней произвольный код.

Глава 7. Хеширование с ключом

Хеш-функции из главы 6 принимают сообщение и возвращают значение его хеша - обычно короткую строку длиной 256 или 512 битов. Любой может вычислить значение хеша сообщения и проверить, что конкретное сообщение хешируется в конкретное значение. Однако, если требуется, чтобы хеши могли вычислять только определенные люди, хеширование выполняют с секретными ключами посредством хеш-функций с ключом.

Хеширование с ключом лежит в основе двух типов криптографических алгоритмов: кодов аутентификации сообщений (message authentication codes, MAC), которые аутентифицируют сообщение и защищают его целостность, и псевдослучайных функций (pseudorandom functions, PRF), формирующих выглядящие случайными значения длиной с хеш. В первом разделе этой главы мы рассмотрим сходства между MAC и PRF, а затем разберемся, как они работают. Одни MAC и PRF основаны на хеш-функциях, другие - на блочных шифрах, а третьи представляют собой оригинальные конструкции. Наконец, мы обсудим примеры атак на в остальном безопасные MAC.

Коды аутентификации сообщений

MAC защищает целостность и подлинность сообщения, создавая значение `T=MAC(K,M)`, которое называется тегом аутентификации сообщения `M`, хотя его часто не вполне точно называют MAC сообщения `M`. Подобно тому как знание ключа шифра позволяет расшифровать сообщение, знание ключа MAC позволяет проверить, что сообщение не было изменено.

Предположим, Алекс и Билл совместно используют ключ `K`, и Алекс отправляет Биллу сообщение `M` вместе с его тегом аутентификации `T=MAC(K,M)`. Получив сообщение и тег, Билл заново вычисляет `MAC(K,M)` и проверяет, равен ли результат полученному тегу. Поскольку вычислить это значение мог только Алекс, Билл знает, что сообщение не было случайно или намеренно повреждено при передаче, то есть подтверждает его целостность, и что сообщение отправил Алекс, то есть подтверждает его подлинность.

MAC в защищенной связи

Системы защищенной связи часто объединяют шифр и MAC, чтобы обеспечить конфиденциальность, целостность и подлинность сообщения. Например, протоколы Internet Protocol Security (IPsec), SSH и TLS создают MAC для каждого передаваемого сетевого пакета.

MAC используются не во всех системах связи. Иногда тег аутентификации создает неприемлемые накладные расходы для каждого пакета - обычно от 64 до 128 битов. Например, старый стандарт мобильной связи GSM шифровал пакеты с закодированными голосовыми вызовами, но не аутентифицировал их. Атакующий мог изменить зашифрованный звуковой сигнал, а получатель этого не заметил бы.

Подделка и атаки с выбранным сообщением

Что означает безопасность MAC? Во-первых, как и у шифра, секретный ключ должен оставаться секретным. Если MAC безопасен, атакующий, не знающий ключа, не должен быть способен создать тег некоторого сообщения. Сфабрикованная пара «сообщение/тег» называется подделкой, а восстановление ключа - частный случай более общего класса атак с подделкой. Свойство безопасности, согласно которому подделку невозможно найти, называется неподделываемостью. Восстановить секретный ключ по списку тегов также должно быть невозможно, иначе атакующие смогут подделывать теги с помощью ключа.

Что может сделать атакующий, пытающийся взломать MAC? Иными словами, какова модель атаки? Самая базовая модель - атака с известным сообщением, при которой пассивно собираются сообщения и соответствующие им теги, например путем прослушивания сети. Но реальные злоумышленники часто могут проводить активные атаки: выбирать сообщения для аутентификации и тем самым получать MAC нужного сообщения. Поэтому стандартной моделью являются атаки с выбранным сообщением, при которых атакующий получает теги для сообщений по своему выбору.

Атаки повторного воспроизведения

MAC не защищают от атак, связанных с повторным воспроизведением тегов. Например, подслушивая связь Алекса и Билла, можно перехватить отправленные Алексом сообщение и тег, а позже снова передать их Биллу, выдавая себя за Алекса. Для предотвращения таких атак повторного воспроизведения протоколы включают в каждое сообщение его номер. Этот номер увеличивается для каждого нового сообщения и аутентифицируется MAC вместе с самим сообщением. Принимающая сторона получает сообщения с номерами 1, 2, 3, 4 и так далее. Поэтому, если атакующий попытается снова отправить сообщение 1, получатель заметит нарушение порядка и возможное повторное воспроизведение предыдущего сообщения 1.

Псевдослучайные функции

PRF использует секретный ключ и возвращает `PRF(K,M)`, причем результат выглядит случайным. Поскольку ключ секретен, выходные значения непредсказуемы для атакующего.

В отличие от MAC, PRF предназначены не для самостоятельного использования, а для включения в криптографический алгоритм или протокол. Например, с помощью PRF можно создавать блочные шифры внутри конструкции Фейстеля; см. раздел «Как строятся блочные шифры» главы 4. Схемы получения ключей применяют PRF для генерации криптографических ключей из мастер-ключа или пароля, а схемы идентификации - для создания ответа на случайный вызов. В сущности, сервер отправляет случайное сообщение-вызов `M`, а клиент в ответ возвращает `PRF(K,M)`, доказывая знание `K`. Стандарт мобильной связи 5G использует PRF для взаимной аутентификации SIM-карты и поставщика услуг, а похожая PRF формирует ключ шифрования и ключ MAC для телефонного разговора. Протокол TLS использует PRF для формирования ключевого материала из главного секрета и случайных значений конкретного сеанса. PRF есть даже во встроенной некриптографической функции `hash()` языка Python, используемой для сравнения объектов.

Безопасность PRF

Безопасная псевдослучайная функция не должна иметь закономерностей, отличающих ее выходные данные от истинно случайных значений. Атакующий, не знающий ключа `K`, не должен уметь отличить результаты `PRF(K,M)` от случайных значений. Иначе говоря, у атакующего не должно быть способа понять, взаимодействует он с алгоритмом PRF или со случайной функцией. Научное название этого свойства безопасности - «неразличимость со случайной функцией». Подробнее о теоретических основах PRF см. том 1, раздел 3.6 книги Голдрайха «Foundations of Cryptography».

PRF сильнее MAC

PRF и MAC являются хешами с ключом, но PRF принципиально сильнее MAC, потому что требования безопасности к MAC слабее. MAC считается безопасным, если атакующий не способен подделывать теги, то есть угадывать выходные данные MAC, тогда как PRF безопасна только в том случае, если ее выходные данные неотличимы от случайных строк. Если выходные данные PRF невозможно отличить от случайных строк, их значения невозможно угадать. Иными словами, любая безопасная PRF является также безопасным MAC.

Обратное утверждение неверно: безопасный MAC не обязательно является безопасной PRF. Предположим, имеется безопасная функция `PRF_1`, на основе которой требуется построить вторую функцию `PRF_2` следующим образом:

```
PRF_2(K,M)=PRF_1(K,M) || 0.
```

Поскольку выход `PRF_2` определяется как выход `PRF_1` с приписанным нулевым битом, он не выглядит столь же случайным, как истинно случайная строка: выходные данные можно отличить по последнему нулевому биту. Следовательно, `PRF_2` не является безопасной PRF. Однако благодаря безопасности `PRF_1` функция `PRF_2` все равно будет безопасным MAC. Если бы вы могли подделать тег `T=PRF_2(K,M)` для некоторого `M`, вы могли бы подделать и тег `PRF_1`, а это, как известно из исходного предположения, невозможно, поскольку `PRF_1` - безопасный MAC. Таким образом, `PRF_2` представляет собой хеш с ключом, являющийся безопасным MAC, но не безопасной PRF.

Но не волнуйтесь: в реальных приложениях такие конструкции MAC не встречаются. В действительности многие развернутые или стандартизированные MAC являются также безопасными PRF и часто используются в обеих ролях. Например, TLS применяет алгоритм HMAC-SHA-256 и как MAC, и как PRF.

Как создавать хеши с ключом из хешей без ключа

На протяжении истории криптографии MAC и PRF редко разрабатывали с нуля: обычно их строили из существующих алгоритмов, чаще всего хеш-функций или блочных шифров. Может показаться очевидным, что хеш-функцию с ключом можно получить, передав обычной хеш-функции ключ и сообщение, но это легче сказать, чем сделать.

Конструкция с секретным префиксом

Первый рассматриваемый метод - конструкция с секретным префиксом - превращает обычную хеш-функцию в хеш с ключом, приписывая ключ перед сообщением и возвращая `Hash(K || M)`. Этот подход небезопасен, когда хеш-функция уязвима к атакам с удлинением сообщения, рассмотренным в главе 6, и когда хеш поддерживает ключи разной длины.

Уязвимость к атакам с удлинением сообщения

Напомним из главы 6, что хеш-функции семейства SHA-2 позволяют атакующим вычислить хеш частично неизвестного сообщения, если известен хеш более короткой версии этого сообщения. Формально атака с удлинением сообщения позволяет вычислить `Hash(K || M_1 || M_2)`, зная только `Hash(K || M_1)` и не зная ни `M_1`, ни `K`. Эти функции позволяют атакующим бесплатно подделывать действительные теги MAC, поскольку они не должны уметь угадывать MAC сообщения `M_1 || M_2`, имея только MAC сообщения `M_1`. Поэтому конструкция с секретным префиксом небезопасна как MAC и PRF при использовании, например, с SHA-256 или SHA-512. Уязвимость к атакам с удлинением сообщения - слабость конструкции Меркла - Дамгора; ни у одного финалиста SHA-3 ее нет. Способность противостоять таким атакам была обязательным требованием к кандидатам SHA-3, см. главу 6.

Уязвимость при ключах разной длины

Конструкция с секретным префиксом также небезопасна, если разрешены ключи разной длины. Например, если ключ `K` - 24-битная шестнадцатеричная строка `123abc`, а `M` равно `def00`, функция `Hash()` обрабатывает значение `K || M=123abcdef00`. Если же `K` - 16-битная строка `123a`, а `M` равно `bcdef000`, функция `Hash()` также обрабатывает `K || M=123abcdef00`. Поэтому результат конструкции с секретным префиксом `Hash(K || M)` для обоих ключей одинаков.

Эта проблема не зависит от лежащего в основе хеша. Ее можно исправить, хешируя длину ключа вместе с ключом и сообщением: например, закодировать битовую длину ключа 16-битным целым числом `L`, а затем вычислить `Hash(L || K || M)`. Однако делать это самостоятельно не следует: современные хеш-функции, такие как BLAKE2 и SHA-3, включают режим с ключом, лишенный этих недостатков и формирующий безопасную PRF, а значит, и безопасный MAC.

Конструкция с секретным суффиксом

Вместо того чтобы хешировать ключ перед сообщением, как в конструкции с секретным префиксом, его можно хешировать после сообщения. Именно так работает конструкция с секретным суффиксом, строящая PRF из хеш-функции как `Hash(M || K)`.

Размещение ключа в конце существенно меняет ситуацию. Атака с удлинением сообщения, работающая против MAC с секретным префиксом, не работает против секретного суффикса. Применение удлинения к MAC с секретным суффиксом дает `Hash(M_1 || K || M_2)` из `Hash(M_1 || K)`, но это не действительная атака, поскольку `Hash(M_1 || K || M_2)` не является допустимым MAC с секретным суффиксом: ключ должен находиться в конце.

Однако конструкция с секретным суффиксом слабее против другого типа атак. Предположим, имеется коллизия хеша `Hash(M_1)=Hash(M_2)` для двух разных сообщений `M_1` и `M_2`, возможно, разной длины. Для хеш-функции M-D, такой как SHA-256, это означает также равенство `Hash(M_1`

|| P_1 || K) и Hash(M_2 || P_2 || K), где P_1 и P_2 - данные дополнения до полного блока. После обработки M_1 || P_1 состояние хеш-функции совпадает с состоянием после обработки M_2 || P_2. Добавление K к каждому экземпляру сохраняет равенство состояний и приводит к коллизии независимо от значения K.

Чтобы воспользоваться этим свойством, атакующий:

1. Находит два коллидирующих сообщения M_1 и M_2.
2. Запрашивает тег MAC сообщения M_1: Hash(M_1 || K).
3. Предполагает, что Hash(M_2 || K) совпадает с ним, тем самым подделывая действительный тег и нарушая безопасность MAC.

Конструкция HMAC

Конструкция hash-based MAC (HMAC) позволяет построить MAC из хеш-функции и безопаснее как секретного префикса, так и секретного суффикса. HMAC дает безопасную PRF, пока лежащий в ее основе хеш устойчив к коллизиям. Но даже если это не так, HMAC все равно дает безопасную PRF при условии, что функция сжатия хеша является PRF. HMAC применялся в протоколах защищенной связи IPsec, SSH и TLS. Спецификации HMAC находятся в стандарте NIST FIPS 198-1 и RFC 2104.

HMAC использует хеш-функцию Hash для вычисления тега MAC, как показано на рисунке 7-1 и в следующем выражении:

```
Hash((K xor opad) || Hash((K xor ipad) || M)).
```

Значение opad, внешнее дополнение, представляет собой шестнадцатеричную строку 5c5c5c...5c длиной с блок функции Hash. Ключ K обычно короче одного блока; его дополняют байтами 00 и складывают посредством XOR с opad. Например, если K - однобайтовая строка 00, то K xor opad=opad. То же верно, если K является строкой из одних нулей любой длины, не превышающей длину блока. K xor opad - первый блок, обрабатываемый внешним вызовом Hash, то есть крайним левым Hash в приведенном выражении или нижним хешем на рисунке 7-1.

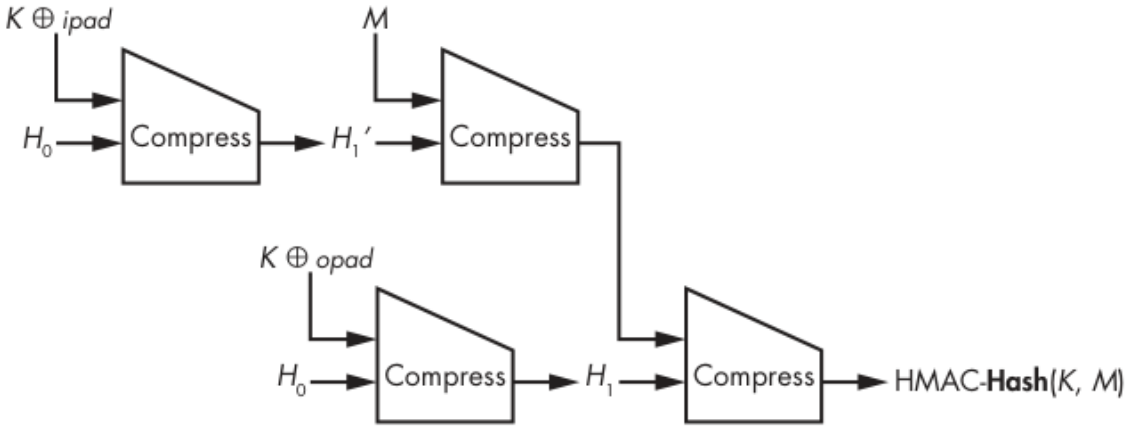

Рисунок 7-1. Конструкция MAC на основе хеша HMAC для хеш-функции, основанной на функции сжатия

Значение ipad, внутреннее дополнение, представляет собой строку 363636...36 длиной с блок функции Hash и дополняется байтами 00. Полученный блок первым обрабатывается внутренним вызовом Hash, то есть крайним правым Hash в выражении или верхним хешем на рисунке 7-1.

Примечание. Метод конверта безопаснее конструкции с секретным префиксом или секретным суффиксом. Он выражается как `Hash(K || M || K)` и представляет собой MAC-сэндвич, но теоретически уступает HMAC по безопасности.

Если в качестве `Hash` используется хеш-функция SHA-256, экземпляр HMAC называется HMAC-SHA-256. В общем случае экземпляр HMAC с хеш-функцией `Hash` называют HMAC-Hash. Поэтому, если кто-то просит вас использовать HMAC, всегда следует спросить: «С какой хеш-функцией?»

Универсальная атака на MAC на основе хеша

Существует атака, работающая против всех MAC на основе итеративной хеш-функции. Вспомните атаку на конструкцию с секретным суффиксом, где коллизия хеша использовалась для получения коллизии MAC. Та же стратегия применима против MAC с секретным префиксом или HMAC, хотя последствия менее разрушительны.

На рисунке 7-2 показан MAC с секретным префиксом `Hash(K || M)`.

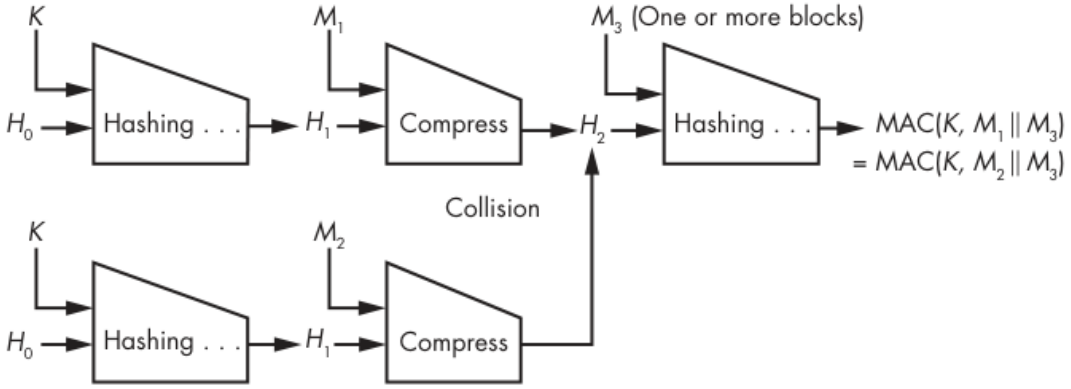

Рисунок 7-2. Принцип универсальной атаки с подделкой на MAC на основе хеша

Если длина дайджеста равна `n` битам, можно найти два сообщения `M_1` и `M_2`, такие что `Hash(K || M_1)=Hash(K || M_2)`, запросив примерно `2^(n/2)` тегов MAC у системы, хранящей ключ. Вспомните атаку дней рождения из главы 6. Если хеш допускает удлинение сообщения, как SHA-256, затем можно использовать `M_1` и `M_2` для подделки MAC: выбрать произвольные данные `M_3` и запросить у оракула MAC значение `Hash(K || M_1 || M_3)`, то есть MAC сообщения `M_1 || M_3`. Оказывается, это также MAC сообщения `M_2 || M_3`, поскольку внутреннее состояние хеша для `M_1` с `M_3` совпадает с состоянием для `M_2` с `M_3`. Тег MAC успешно подделан. При `n` больше, например, 128 битов требуемые усилия становятся неосуществимыми.

Эта атака работает, даже если хеш-функция не уязвима к удлинению сообщения, и применима также к HMAC. Достаточно внутренней коллизии хеш-функции, необязательно коллизии полного хеша. Стоимость атаки зависит как от размера цепочного значения, так и от длины MAC: если цепочное значение MAC имеет длину 512 битов, а теги - 128 битов, вычисление за `2^64` операций найдет коллизию MAC, но, скорее всего, не коллизию внутреннего состояния, поскольку для ее нахождения требуется в среднем `2^(512/2)=2^256` операций.

Как создавать хеши с ключом из блочных шифров

Функции сжатия во многих хеш-функциях строятся на блочных шифрах, см. главу 6. Например, PRF HMAC-SHA-256 представляет собой последовательность вызовов функции сжатия SHA-256, которая является блочным шифром, повторяющим последовательность раундов. Иными словами, HMAC-SHA-256 - это блочный шифр внутри функции сжатия внутри хеша внутри конструкции HMAC. Так почему бы не использовать блочный шифр напрямую вместо такой многослойной конструкции?

Cipher-based MAC (CMAC) - именно такая конструкция: она создает MAC, имея только блочный шифр, например AES. Хотя CMAC менее популярен, чем HMAC, он развернут во множестве систем, включая протокол Internet Key Exchange (IKE), входящий в набор IPsec. Например, IKE генерирует ключевой материал, используя в качестве основного алгоритма конструкцию AES-CMAC-PRF-128, то есть основанный на AES CMAC со 128-битным выходом. CMAC определен в RFC 4493.

Взлом CBC-MAC

CMAC был разработан в 2005 году как улучшенная версия CBC-MAC - более простого MAC на основе блочного шифра, производного от режима работы блочного шифра с сцеплением блоков шифртекста (CBC), рассмотренного в главе 4.

CBC-MAC, предшественник CMAC, прост: чтобы вычислить тег сообщения `M` при помощи блочного шифра `E`, нужно зашифровать `M` в режиме CBC с нулевым начальным значением (IV) и отбросить все блоки шифртекста, кроме последнего. Иными словами, вычисляется

```
C_1=E(K,M_1), C_2=E(K,M_2 xor C_1), C_3=E(K,M_3 xor C_2)
```

и так далее для каждого блока `M`, после чего сохраняется только последний `C_i` - тег CBC-MAC сообщения `M`. Просто и так же просто атакуется.

Чтобы понять, почему CBC-MAC небезопасен, рассмотрим тег CBC-MAC `T_1=E(K,M_1)` одноблочного сообщения `M_1` и тег `T_2=E(K,M_2)` другого одноблочного сообщения `M_2`. Имея две пары `(M_1,T_1)` и `(M_2,T_2)`, можно заключить, что `T_2` является также тегом двухблочного сообщения `M_1 || (M_2 xor T_1)`. Если применить CBC-MAC к `M_1 || (M_2 xor T_1)`, получится `C_1=E(K,M_1)=T_1`, а затем `C_2=E(K,(M_2 xor T_1) xor T_1)=E(K,M_2)=T_2`. Таким образом, из двух пар «сообщение/тег» можно создать третью, не зная ключа. Иными словами, можно подделывать теги CBC-MAC, нарушая его безопасность.

Исправление CBC-MAC

CMAC исправляет CBC-MAC, обрабатывая последний блок ключом, отличным от ключа предыдущих блоков. Для этого CMAC сначала получает из основного ключа `K` два ключа `K_1` и `K_2`, причем `K`, `K_1` и `K_2` различны. Последний блок CMAC обрабатывает либо с `K_1`, либо с `K_2`, тогда как предыдущие блоки используют `K`.

Чтобы определить `K_1` и `K_2`, CMAC сначала вычисляет временное значение `L=E(0,K)`, где 0 служит ключом блочного шифра, а `K` - блоком открытого текста. Затем CMAC полагает `K_1=(L<<1)`, если старший значащий бит (most significant bit, MSB) `L` равен 0, и `K_1=(L<<1) xor 87`, если MSB `L` равен 1. Шестнадцатеричное число 87, равное 135 в десятичной системе, выбрано за математические свойства для 128-битных блоков данных; если размер блока отличается от 128 битов, требуется другое значение. Ключ `K_2` полагается равным `(K_1<<1)`, если MSB `K_1` равен 0, и

K_2=(K_1<<1) xor 87 в противном случае.

Имея K_1 и K_2, CMAC работает как CBC-MAC, за исключением последнего блока. Если последний фрагмент сообщения M_n имеет ровно размер блока, CMAC возвращает в качестве тега E(K,M_n xor C_(n-1) xor K_1), как показано на рисунке 7-3.

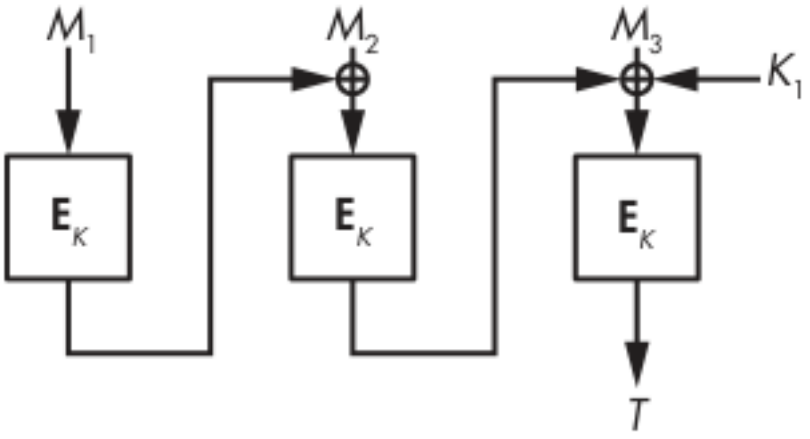

Рисунок 7-3. Конструкция MAC на основе блочного шифра CMAC, когда сообщение состоит из целого числа блоков

Если в M_n меньше битов, чем в блоке, CMAC дополняет его единичным битом и нулями и возвращает в качестве тега E(K,M_n xor C_(n-1) xor K_2), как показано на рисунке 7-4.

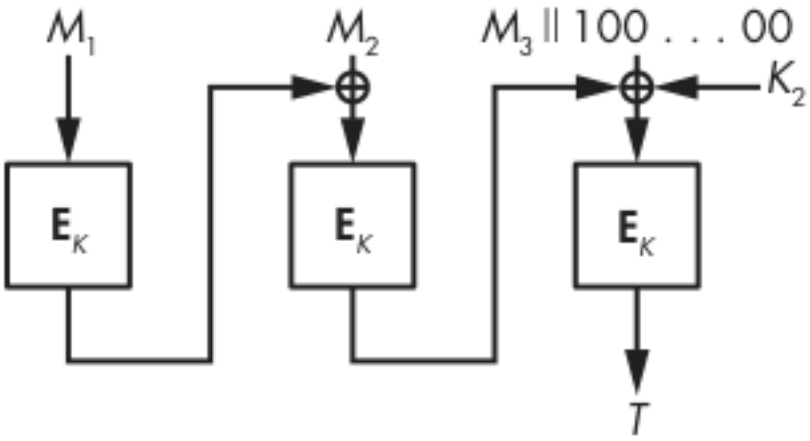

Рисунок 7-4. Конструкция MAC на основе блочного шифра CMAC, когда последний блок сообщения дополняется единичным битом и нулями до полного блока

В первом случае используется только K_1, а во втором - только K_2, но в обоих случаях фрагменты сообщения перед последним обрабатываются основным ключом K.

В отличие от режима шифрования CBC, CMAC не принимает IV как параметр и является детерминированным: при одном ключе CMAC всегда возвращает один и тот же тег заданного сообщения M, поскольку вычисление CMAC(M) не рандомизировано. Это допустимо: в отличие от шифрования, вычисление MAC не обязано быть рандомизированным для безопасности, что избавляет от необходимости выбирать случайный IV.

Специализированные конструкции MAC

Вы увидели, как повторно использовать хеш-функции и блочные шифры для построения PRF, остающихся безопасными, пока безопасен лежащий в основе хеш или шифр. Такие схемы, как HMAC и CMAC, просто объединяют доступные хеш-функции или блочные шифры, получая безопасную PRF или MAC. Повторное использование готовых алгоритмов удобно, но действительно ли это самый эффективный подход?

Интуитивно для безопасности PRF и MAC должно требоваться меньше работы, чем для хеш-функций без ключа: секретный ключ не позволяет атакующим экспериментировать с алгоритмом, поскольку у них нет ключа. Кроме того, PRF и MAC открывают атакующим только короткий тег, в отличие от блочных шифров, открывающих шифртекст длиной с сообщение. Поэтому PRF и MAC не должна требоваться вся мощь хеш-функций или блочных шифров. В этом смысл специализированной конструкции, то есть алгоритмов, создаваемых исключительно для роли PRF или MAC.

Следующие разделы посвящены двум таким алгоритмам: Poly1305 и SipHash. Я объясню принципы их устройства и причины их предполагаемой безопасности.

Poly1305

Алгоритм Poly1305, произносимый «поли-тринадцать-ноль-пять», был разработан в 2005 году Дэниелом Дж. Бернстайном, создателем потокового шифра Salsa20 из главы 5 и шифра ChaCha, вдохновившего хеш-функции BLAKE и BLAKE2 из главы 6. Poly1305 оптимизирован для сверхбыстрой работы на современных CPU и на момент написания книги входит в число алгоритмов, поддерживаемых TLS 1.3 и OpenSSH, а также множеством других приложений. В отличие от Salsa20, конструкция Poly1305 основана на методах 1970-х годов: универсальных хеш-функциях и конструкции Вегмана - Картера.

Универсальные хеш-функции

MAC Poly1305 использует универсальную хеш-функцию. Такая хеш-функция слабее криптографической, но намного быстрее. Например, универсальные хеш-функции не обязаны быть устойчивыми к коллизиям.

Подобно PRF, универсальный хеш параметризуется секретным ключом: результат универсальной хеш-функции `UH` для сообщения `M` и ключа `K` будем записывать как `UH(K,M)`. У универсальной хеш-функции только одно требование безопасности: для любых двух сообщений `M_1` и `M_2` и случайного ключа `K` вероятность равенства `UH(K,M_1)` и `UH(K,M_2)` должна быть пренебрежимо мала. В отличие от PRF, универсальный хеш не обязан быть псевдослучайным; просто не должно существовать пары `(M_1,M_2)`, дающей один хеш для множества разных ключей. Поскольку их требования безопасности выполнить легче, универсальные хеш-функции требуют меньше операций и намного быстрее PRF.

Однако универсальный хеш можно использовать как MAC для аутентификации не более одного сообщения. В качестве примера рассмотрим универсальный хеш с вычислением многочлена, применяемый в Poly1305. См. основополагающую статью Эдгара Гилберта, Джесси Мак-Уильямс и Нила Слоуна «Codes Which Detect Deception» 1974 года, где это понятие описано подробнее. Такой хеш с вычислением многочлена параметризуется простым числом `p` и принимает ключ из двух чисел `R` и `K` в диапазоне `[1,p)` и сообщение `M`, состоящее из `n` блоков `(M_1,M_2,...,M_n)`. Результат универсального хеша вычисляется следующим образом:

```
UH(R,K,M)=(R+M_1K+M_2K^2+M_3K^3+...+M_nK^n) mod p.
```

В этом уравнении знак плюс обозначает сложение положительных целых чисел, `K^i` - число `K` в степени `i`, а `mod p` - приведение результата по модулю `p`, то есть остаток от деления результата на `p`. Например, `12 mod 10=2`, `10 mod 10=0`, `8 mod 10=8` и так далее.

Поскольку хеширование должно выполняться как можно быстрее, MAC на основе универсального хеша часто работают со 128-битными блоками сообщений и простым числом `p`, немного превышающим `2^128`, например `2^128+51`. Ширина 128 битов позволяет создавать очень быстрые реализации, эффективно использующие 32- и 64-битные арифметические устройства распространенных CPU.

Ограничения безопасности

У универсальных хешей есть одна слабость: поскольку универсальный хеш безопасно аутентифицирует лишь одно сообщение, атакующий может взломать приведенный выше MAC с вычислением многочлена, запросив теги всего двух сообщений. В частности, он может запросить тег сообщения, все блоки которого равны нулю, `M_1=M_2=...=0`, и получить `UH(R,K,0)=R`, тем самым найдя секретное значение `R`. Затем он может запросить тег сообщения, в котором `M_1=1`, а `M_2=M_3=...=0`; его тег равен `T=R+K`. Вычтя `R` из `T`, атакующий найдет `K`. Теперь он знает весь ключ `(R,K)` и способен подделать MAC любого сообщения.

К счастью, существует способ перейти от безопасности одного сообщения к безопасности множества сообщений.

MAC Вегмана - Картера

Прием аутентификации множества сообщений с помощью универсальной хеш-функции появился благодаря исследователям IBM Марку Вегману и Лоуренсу Картеру и их статье 1981 года «New Hash Functions and Their Use in Authentication and Set Equality». Конструкция Вегмана - Картера строит MAC из универсальной хеш-функции и PRF, использует два ключа `K_1` и `K_2` и возвращает

```
MAC(K_1,K_2,M,N)=UH(K_1,M)+PRF(K_2,N),
```

где `N` - одноразовый номер, который должен использоваться не более одного раза для каждого ключа `K_2`, а выход PRF имеет тот же размер, что выход универсальной хеш-функции UH. При сложении этих значений сильный псевдослучайный выход PRF маскирует криптографическую слабость UH. Это можно рассматривать как шифрование результата универсального хеша, где PRF действует как потоковый шифр и предотвращает описанную выше атаку, позволяя аутентифицировать множество сообщений одним ключом `K_1`.

Подведем итог: конструкция Вегмана - Картера `UH(K_1,M)+PRF(K_2,N)` дает безопасный MAC при следующих предположениях:

• UH является безопасным универсальным хешем.
• PRF является безопасной PRF.
• Каждый одноразовый номер `N` используется только один раз для каждого ключа `K_2`.
• Выходные значения UH и PRF достаточно длинны для обеспечения требуемого уровня безопасности.

Теперь рассмотрим, как Poly1305 использует конструкцию Вегмана - Картера для построения безопасного и быстрого MAC.

Poly1305-AES

Изначально Poly1305 был предложен как Poly1305-AES, объединяющий универсальный хеш Poly1305 с блочным шифром AES. Poly1305-AES намного быстрее MAC на основе HMAC и даже CMAC, поскольку вычисляет всего один блок AES, а сообщение обрабатывает параллельно с помощью ряда простых арифметических операций.

Для 128-битных `K_1`, `K_2` и `N` и сообщения `M` Poly1305-AES возвращает

```
(Poly1305(K_1,M)+AES(K_2,N)) mod 2^128.
```

Приведение по модулю `2^128` гарантирует, что результат помещается в 128 битов. Poly1305 разбирает сообщение `M` как последовательность 128-битных блоков `(M_1,M_2,...,M_n)` и добавляет к старшему разряду каждого блока 129-й бит, делая все блоки 129-битными. Если последний блок короче 16 байтов, перед добавлением последнего 129-го бита он дополняется единичным битом, за которым следуют нулевые. Затем Poly1305 вычисляет многочлен:

```
Poly1305(K_1,M)=(M_1K_1^n+M_2K_1^(n-1)+...+M_nK_1) mod (2^130-5).
```

Результат этого выражения - целое число длиной не более 129 битов. Когда к нему прибавляется 128-битное значение `AES(K_2,N)`, результат приводится по модулю `2^128` и образует 128-битный MAC.

> **Примечание.** Я описал конструкцию Вегмана - Картера как использующую PRF, однако AES - не PRF, а псевдослучайная перестановка (pseudorandom permutation, PRP). Но это несущественно, поскольку конструкция Вегмана - Картера хорошо работает и с PRP, и с PRF. Если дана функция, являющаяся либо PRF, либо PRP, по одним ее выходным значениям трудно определить, к какому типу она относится. Иными словами, отличить PRP от PRF вычислительно трудно.

Анализ безопасности Poly1305-AES, см. статью «The Poly1305-AES Message-Authentication Code», показывает, что Poly1305-AES обеспечивает 128-битную безопасность, пока AES остается безопасным блочным шифром и все реализовано правильно, как и должно быть для любого криптографического алгоритма.

Универсальный хеш Poly1305 можно объединять не только с AES. Например, Poly1305 применялся с потоковым шифром ChaCha, см. RFC 7539 «ChaCha20 and Poly1305 for IETF Protocols». Нет сомнений, что Poly1305 продолжат использовать везде, где требуется быстрый MAC.

Хотя Poly1305 быстр и безопасен, у него есть несколько недостатков. Во-первых, вычисление многочлена трудно реализовать эффективно, особенно тем, кто не знаком с соответствующими математическими понятиями; примеры см. на https://github.com/floodyberry/poly1305-donna. Во-вторых, сам по себе он безопасен лишь для одного сообщения, если не используется конструкция Вегмана - Картера. Но в таком случае ему нужен одноразовый номер, и при его повторении алгоритм становится небезопасным. Наконец, Poly1305 оптимизирован для длинных сообщений, но избыточен при обработке только коротких, например короче 128 байтов. В таких случаях хорошим решением является SipHash.

SipHash

Я разработал SipHash в 2012 году вместе с Дэном Бернстайном, первоначально для решения некриптографической проблемы: атак отказа в обслуживании на хеш-таблицы. Хеш-таблицы - это структуры данных, широко применяемые, в частности, в языках программирования для эффективного внутреннего хранения элементов. До SipHash хеш-таблицы использовали некриптографические хеш-функции с ключом, коллизии для которых часто было легко найти; это позволяло замедлять систему и проводить атаки отказа в обслуживании. Мы заметили, что PRF решит эту проблему, и приступили к разработке SipHash - PRF, подходящей для хеш-таблиц. Поскольку хеш-таблицы в основном обрабатывают короткие входные данные, SipHash оптимизирована для коротких сообщений. SipHash - полноценная PRF и MAC, особенно эффективная там, где большинство входных данных короткие.

SipHash использует прием, делающий ее безопаснее базовых губчатых функций: вместо однократного XOR блока сообщения с состоянием перед перестановкой SipHash выполняет XOR до и после перестановки, как показано на рисунке 7-5. 128-битный ключ SipHash рассматривается как два 64-битных слова `K_1` и `K_2`, складываемых посредством XOR с фиксированным 256-битным начальным состоянием, которое рассматривается как четыре 64-битных слова. После этого ключи отбрасываются, и вычисление SipHash сводится к итеративному применению основной функции SipRound и XOR фрагментов сообщения для изменения внутреннего состояния из четырех слов. Наконец, SipHash возвращает 64-битный тег, складывая посредством XOR четыре слова состояния.

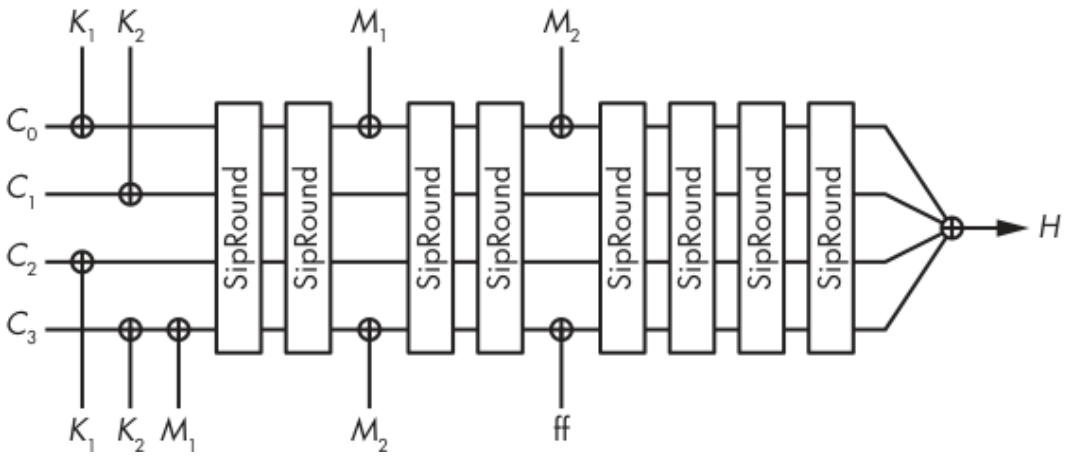

Рисунок 7-5. Обработка 15-байтового сообщения функцией SipHash-2-4: блок `M_1` из 8 байтов, блок `M_2` из 7 байтов и 1 байт дополнения

Функция SipRound использует множество XOR вместе со сложениями и циклическими сдвигами слов для обеспечения безопасности. SipRound преобразует состояние из четырех 64-битных слов `(a,b,c,d)`, выполняя следующие операции сверху вниз. Операции слева и справа независимы и могут выполняться параллельно:

```
a += b          c += d
b <<<= 13       d <<<= 16
b ^= a          d ^= c
a <<<= 32
c += b          a += d
b <<<= 17       d <<<= 21
b ^= c          d ^= a
c <<<= 32
```

Здесь `a += b` - сокращенная запись `a = a + b`, а `b <<<= 13` - сокращенная запись `b = b <<< 13`, то есть 64-битное слово `b` циклически сдвигается на 13 битов влево.

Эти простые операции над 64-битными словами - почти все, что нужно реализовать для вычисления SipHash, хотя самостоятельно реализовывать ее не придется. Готовые реализации доступны для большинства языков, включая C, Go, Java, JavaScript, Python и Rust.

> **Примечание.** Обозначение SipHash-x-y означает версию SipHash, выполняющую x раундов SipRound между каждым вводом блока сообщения, а затем y раундов. Большее число раундов требует больше операций и замедляет работу, но повышает безопасность. Версия по умолчанию - SipHash-2-4, обычно обозначаемая просто SipHash; пока она успешно противостоит криптоанализу. Я также определил SipHash128 - версию SipHash со 128-битными тегами.

Многие системы, в том числе язык Rust, операционная система OpenBSD и блокчейн Bitcoin, используют SipHash внутри. Ядро Linux тоже использует SipHash, а также HalfSipHash, «небезопасного младшего родственника SipHash», - уменьшенную версию с 64-битным ключом и 32-битным выходом; см. https://docs.kernel.org/security/siphash.html.

Что может пойти не так

Подобно шифрам и хеш-функциям без ключа, MAC и PRF, безопасные на бумаге, могут оказаться уязвимыми к атакам в реальной среде. Рассмотрим два примера.

Атаки по времени на проверку MAC

Атаки по побочным каналам направлены не на сам криптографический алгоритм, а на его реализацию. В частности, атаки по времени используют длительность выполнения алгоритма для определения секретных данных: ключей, открытого текста и секретных случайных значений. Сравнение строк за переменное время создает уязвимости не только при проверке MAC, но и во множестве других криптографических функций и средств безопасности.

MAC могут быть уязвимы к атакам по времени, если удаленная система проверяет теги за время, зависящее от значения тега. Это позволяет атакующему определить правильный тег сообщения, испытывая множество неправильных и выявляя тот, проверка которого занимает больше всего времени. Проблема возникает, когда сервер сравнивает правильный тег с неправильным, последовательно сопоставляя две строки байт за байтом до первого различия. Например, код Python в листинге 7-1 сравнивает две строки байт за байтом за переменное время: если первые байты различаются, функция возвращает результат после одного сравнения; если строки x и y совпадают, функция выполняет n сравнений по длине строк.

```
def compare_mac(x, y, n):
    if len(x) != len(y):
        return False
    if len(x) != n:
        return False
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True
```

Листинг 7-1. Сравнение двух n-байтовых строк за переменное время

Чтобы продемонстрировать уязвимость функции `compare_mac()`, напишем программу, измеряющую время 100 000 вызовов `compare_mac()`: сначала для одинаковых 16-символьных значений x и y, а затем для значений, различающихся третьим байтом. Второе сравнение должно занимать заметно меньше времени, поскольку `compare_mac()` сравнивает меньше байтов, чем при

одинаковых x и y, как показано в листинге 7-2.

```
from time import time

MAC1 = 'abcdefghijklmnop'
MAC2 = 'abXdefghijklmnop'
TRIALS = 100000

def compare_mac(x, y, n):
    if len(x) != len(y):
        return False
    if len(x) != n:
        return False
    for i in range(n):
        if x[i] != y[i]:
            return False
    return True

# Каждый вызов compare_mac() рассмотрит все 16 символов.
start = time()
for i in range(TRIALS):
    compare_mac(MAC1, MAC1, len(MAC1))
end = time()
print("%0.5f" % (end-start))

# Каждый вызов compare_mac() рассмотрит три символа.
start = time()
for i in range(TRIALS):
    compare_mac(MAC1, MAC2, len(MAC1))
end = time()
print("%0.5f" % (end-start))
```

Листинг 7-2. Измерение различий во времени выполнения `compare_mac()` из листинга 7-1

На MacBook Pro с процессором ARM M1 один запуск программы из листинга 7-2 выводит время выполнения около 67 и 26 миллисекунд соответственно. Разница достаточно велика, чтобы понять, что происходит внутри алгоритма. Перемещая различающийся байт в другие позиции строки, можно наблюдать разное время выполнения для разных смещений. Если `MAC1` - правильный тег MAC, а `MAC2` - тег, испытываемый атакующим, легко определить положение первого различия, то есть число правильно угаданных байтов.

Если время выполнения не зависит от секретного значения, атака по времени не сработает. Поэтому разработчики стремятся создавать реализации с постоянным временем - код, выполняющийся строго за одно и то же время при любом значении секретных входных данных. Например, функция C в листинге 7-3 сравнивает два буфера длиной `size` байтов за постоянное время: временная переменная `result` отлична от нуля тогда и только тогда, когда два буфера где-либо различаются.

```
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 */
}
```

Листинг 7-3. Сравнение двух буферов за постоянное время для более безопасной проверки MAC

Когда губки дают утечку

Алгоритмы на основе перестановок, такие как SHA-3 и SipHash, просты, легко реализуются и имеют компактные реализации, но уязвимы перед атаками по побочным каналам, восстанавливающими снимок состояния системы. Например, если процесс способен в любой момент прочитать значения оперативной памяти и регистров либо дамп памяти процесса, атакующий может определить внутреннее состояние SHA-3 в режиме MAC или внутреннее состояние SipHash, а затем выполнить обратную перестановку и восстановить исходное секретное состояние. После этого он сможет подделывать теги любых сообщений, нарушив безопасность MAC.

К счастью, такая атака не работает против MAC на основе функций сжатия, таких как HMAC-SHA-256 и BLAKE2 с ключом, поскольку атакующему требуется снимок памяти в точный момент использования ключа. Следовательно, если в среде возможна утечка частей памяти процесса, можно использовать MAC на основе необратимого преобразования функции сжатия, а не перестановки.

Дополнительная литература

Почтенный HMAC заслуживает большего внимания, чем позволяет объем этой главы, особенно цепочка идей, приведшая к его широкому распространению и последующему краху при объединении со слабой хеш-функцией. Рекомендую статью Михира Белларе, Рана Канетти и Хьюго Кравчика 1996 года «Keying Hash Functions for Message Authentication», в которой были представлены HMAC и родственная конструкция NMAC, а также последующую статью Белларе 2006 года «New Proofs for NMAC and HMAC: Security Without Collision-Resistance». В ней доказано, что HMAC не требуется устойчивый к коллизиям хеш: достаточно хеша с функцией сжатия, являющейся PRF. Что касается атак, в статье Пьера-Алена Фука, Гаэтана Лерена и Фонга Нгуена 2007 года «Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5» показано, как атаковать HMAC и NMAC, построенные поверх хрупкой хеш-функции, такой как MD4 или MD5. HMAC-MD5 и HMAC-SHA-1 не полностью взломаны, но риск достаточно высок.

MAC Вегмана - Картера также заслуживают большего внимания как из-за практической ценности, так и из-за лежащей в их основе теории. Основополагающие статьи Вегмана и Картера доступны на https://cr.yp.to/bib/entries.html. Среди других современных конструкций - UMAC и VMAC, относящиеся к самым быстрым MAC для длинных сообщений.

Один тип MAC, не рассмотренный в этой главе, - Pelican. Он использует блочный шифр AES, сокращенный до четырех раундов вместо десяти в полном шифре, для аутентификации фрагментов сообщений в простой конструкции, описанной на https://eprint.iacr.org/2005/088. Однако Pelican представляет скорее академический интерес и редко применяется на практике.

И последнее: если вас интересует поиск уязвимостей в криптографическом программном обеспечении, ищите случаи применения CBC-MAC или слабости, вызванные обработкой HMAC ключей произвольной длины, когда в качестве ключа берется `Hash(K)` вместо `K`, если `K` слишком длинен, что делает `K` и `Hash(K)` эквивалентными ключами. Или просто ищите системы, не использующие MAC там, где он необходим, - это встречается часто.

В главе 8 мы объединим MAC с шифрами для защиты подлинности, целостности и конфиденциальности сообщения. Кроме того, мы сделаем это без MAC с помощью аутентифицированных шифров, которые объединяют возможности обычного шифра и MAC, возвращая тег вместе с каждым шифртекстом.

Глава 8. Аутентифицированное шифрование

Эта глава посвящена типу алгоритмов, защищающих не только конфиденциальность, но и подлинность сообщения. Напомним из главы 7, что коды аутентификации сообщений (MAC) защищают подлинность сообщения, создавая тег - разновидность подписи. Подобно MAC, рассматриваемые здесь алгоритмы аутентифицированного шифрования (authenticated encryption, AE) формируют тег аутентификации, но при этом еще и шифруют сообщение. Иными словами, один алгоритм AE предоставляет возможности и обычного шифра, и MAC.

Объединение шифра и MAC может давать разные уровни аутентифицированного шифрования, как вы узнаете из этой главы. Мы рассмотрим несколько способов объединения MAC с шифрами, обсудим, какие методы безопаснее всего, и изучим шифры, создающие одновременно шифртекст и тег аутентификации. Затем мы рассмотрим четыре важных аутентифицированных шифра: три конструкции на основе блочных шифров с особым вниманием к популярному Advanced Encryption Standard в режиме счетчика Галуа (AES-GCM) и шифр, использующий только алгоритм перестановки.

Аутентифицированное шифрование с помощью MAC

На рисунке 8-1 показаны три способа объединить MAC и шифры для одновременного шифрования и аутентификации открытого текста: «шифрование и MAC», «MAC, затем шифрование» и «шифрование, затем MAC».

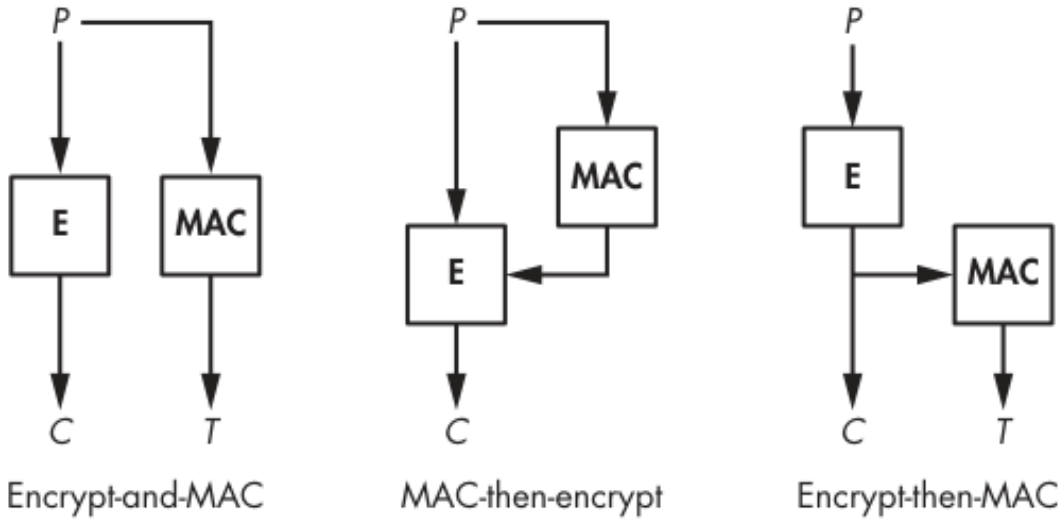

Рисунок 8-1. Комбинации шифра и MAC

Эти комбинации различаются порядком шифрования и формирования тега аутентификации. Выбор конкретного алгоритма MAC или шифра не важен, пока каждый из них безопасен сам по себе, а MAC и шифр используют разные ключи.

В композиции «шифрование и MAC» открытый текст шифруется, а тег аутентификации формируется непосредственно из открытого текста. Две операции - шифрование и аутентификация - независимы и потому могут выполняться параллельно. В схеме «MAC, затем шифрование» сначала из открытого текста создается тег, а затем открытый текст и MAC шифруются вместе. В методе «шифрование, затем MAC» сначала шифруется открытый текст, а потом из шифртекста формируется тег. Посмотрим, какой метод, вероятнее всего, безопаснее.

Подход «шифрование и MAC»

При подходе «шифрование и MAC» шифртекст и тег MAC вычисляются отдельно. Для открытого текста P отправитель вычисляет шифртекст C=E(K_1,P), где E - алгоритм шифрования, а C - получившийся шифртекст. Тег аутентификации T вычисляется из открытого текста как T=MAC(K_2,P). Две операции независимы и потому могут выполняться параллельно.

Сформировав шифртекст и тег аутентификации, отправитель передает оба предполагаемому получателю. Получив C и T, получатель расшифровывает C и получает открытый текст P, вычисляя P=D(K_1,C). Затем он вычисляет MAC(K_2,P) по расшифрованному открытому тексту и сравнивает результат с полученным T. Если C или T повреждены, проверка завершается неудачей и сообщение считается недействительным.

Теоретически «шифрование и MAC» - наименее безопасная композиция MAC и шифра, поскольку даже безопасный MAC может раскрывать сведения о P, облегчая его восстановление. Цель применения MAC состоит лишь в том, чтобы теги нельзя было подделать; теги не обязаны выглядеть случайными. Поэтому тег аутентификации T открытого текста P может раскрывать информацию, хотя MAC считается безопасным. Если MAC является псевдослучайной функцией, тег ничего не раскроет о P.

Несмотря на относительную слабость, многие системы продолжают поддерживать «шифрование и MAC», включая протокол защищенного транспортного уровня SSH. В нем за каждым зашифрованным пакетом C следует тег T=MAC(K,N || P), где N - 32-битный порядковый номер, увеличиваемый для каждого пакета. На практике «шифрование и MAC» оказалось достаточно надежным для SSH благодаря сильным алгоритмам MAC, таким как HMAC-SHA-256, которые не раскрывают сведений о P. Введите следующую команду, чтобы увидеть список MAC, поддерживаемых OpenSSH:

```
$ ssh -Q mac
```

Композиция «MAC, затем шифрование»

Композиция «MAC, затем шифрование» защищает сообщение P, сначала вычисляя тег аутентификации T=MAC(K_2,P). Затем она создает шифртекст, совместно шифруя открытый текст и тег: C=E(K_1,P || T).

После этих действий отправитель передает только C, содержащий и зашифрованный открытый текст, и тег. Получатель расшифровывает C, вычисляя P || T=D(K_1,C), и получает открытый текст и тег T. Затем он проверяет полученный тег T, непосредственно вычисляя MAC(K_2,P) по открытому тексту и подтверждая равенство результата тегу T.

Как и при «шифровании и MAC», при использовании «MAC, затем шифрование» получатель должен расшифровать C, прежде чем сможет определить, поврежден ли пакет. В результате потенциально поврежденный открытый текст попадает к получателю. Тем не менее «MAC, затем шифрование» безопаснее «шифрования и MAC», поскольку скрывает тег аутентификации открытого текста и тем самым не позволяет тегу раскрывать сведения об открытом тексте.

Протокол TLS много лет использовал «MAC, затем шифрование», но в TLS 1.3 эту композицию заменили аутентифицированными шифрами; подробнее о TLS 1.3 см. в главе 13.

Композиция «шифрование, затем MAC»

Композиция «шифрование, затем MAC» отправляет получателю два значения: шифртекст `C=E(K_1,P)` и основанный на нем тег `T=MAC(K_2,C)`. Получатель вычисляет `MAC(K_2,C)` и проверяет равенство полученному `T`. Если значения равны, открытый текст вычисляется как `P=D(K_1,C)`; если нет, шифртекст отбрасывается.

Преимущество этого метода в том, что для обнаружения поврежденных сообщений получателю достаточно вычислить MAC, а расшифровывать поврежденный шифртекст не нужно. Кроме того, атакующий не может отправлять получателю пары `C` и `T` для расшифрования, не взломав MAC, поэтому передать получателю вредоносные данные становится труднее.

Благодаря сочетанию этих свойств «шифрование, затем MAC» сильнее подходов «шифрование и MAC» и «MAC, затем шифрование». Это одна из причин, по которым широко используемый набор протоколов защищенной связи IPsec применяет его для защиты пакетов, например внутри VPN-туннелей.

Обратите внимание, что SSH и TLS не используют «шифрование, затем MAC», поскольку на момент их создания другие подходы казались достаточными, а не потому, что теоретических слабостей не существовало. Нежелательные свойства не обязательно превращаются в реальные уязвимости.

Аутентифицированные шифры

Аутентифицированные шифры - альтернатива комбинациям шифра и MAC. Они похожи на обычные шифры, но возвращают вместе с шифртекстом тег аутентификации.

Шифрование аутентифицированным шифром обозначается `AE(K,P)=(C,T)`. AE означает authenticated encryption, аутентифицированное шифрование, которое на основе ключа `K` и открытого текста `P` возвращает шифртекст `C` и сформированный тег аутентификации `T`. Иными словами, один алгоритм аутентифицированного шифра выполняет ту же работу, что композиция шифра и MAC, причем проще, быстрее и часто безопаснее.

Расшифрование аутентифицированным шифром обозначается `AD(K,C,T)=P`. Здесь AD означает authenticated decryption, аутентифицированное расшифрование, которое по шифртексту `C`, тегу `T` и ключу `K` возвращает открытый текст `P`. Если проверка тега не проходит, AD возвращает ошибку, не позволяя получателю обработать открытый текст, который мог быть подделан. Соответственно, если AD возвращает открытый текст, он был зашифрован человеком или системой, знающими секретный ключ.

Основные требования безопасности к аутентифицированному шифру просты: его аутентификация должна быть столь же сильной, как у MAC. Это означает, что подделать пару «шифртекст/тег» `(C,T)`, которую функция расшифрования AD примет и расшифрует, должно быть невозможно.

Что касается конфиденциальности, аутентифицированный шифр принципиально сильнее обычного, поскольку системы, хранящие секретный ключ, расшифровывают шифртекст только при действительном теге аутентификации. Если тег недействителен, открытый текст отбрасывается. Это свойство не позволяет атакующим выполнять запросы с выбранным шифртекстом - атаку, при которой они создают шифртексты и запрашивают соответствующие им открытые тексты.

Аутентифицированное шифрование со связанными данными

Криптографы называют связанными данными любые данные, обрабатываемые аутентифицированным шифром так, что они аутентифицируются благодаря тегу, но не шифруются. По умолчанию все открытые данные, переданные аутентифицированному шифру, одновременно шифруются и аутентифицируются.

Предположим, нужно аутентифицировать сообщение, включая его незашифрованные части, но зашифровать не все сообщение, то есть передать и аутентифицировать данные в дополнение к зашифрованному сообщению. Например, если шифр обрабатывает сетевой пакет, состоящий из заголовка и полезной нагрузки, полезную нагрузку можно зашифровать, чтобы скрыть передаваемые данные, но заголовок оставить незашифрованным, поскольку он содержит сведения, необходимые для доставки пакета конечному получателю. При этом данные заголовка все равно желательно аутентифицировать, чтобы убедиться, что они получены от ожидаемого отправителя.

Для решения этих задач криптографы создали понятие аутентифицированного шифрования со связанными данными (authenticated encryption with associated data, AEAD). Алгоритм AEAD позволяет присоединить открытые данные к шифртексту так, что при повреждении открытых данных тег аутентификации не пройдет проверку, а шифртекст не будет расшифрован. Такие открытые данные должны кодироваться и сериализоваться безопасным способом, исключающим неоднозначное толкование содержимого.

Операцию AEAD можно записать как `AEAD(K,P,A)=(C,A,T)`. По ключу `K`, открытому тексту `P` и связанным данным `A` функция AEAD возвращает шифртекст, незашифрованные связанные данные `A` и тег аутентификации. AEAD оставляет связанные данные без изменений, а шифртекст представляет собой зашифрованный открытый текст. Тег аутентификации зависит и от `P`, и от `A` и будет признан действительным, только если ни `C`, ни `A` не изменены.

Поскольку тег аутентификации зависит от `A`, расшифрование со связанными данными вычисляется как `ADAD(K,C,A,T)=(P,A)`. Для получения открытого текста и связанных данных расшифрованию нужны ключ, шифртекст, связанные данные и тег; если `C` или `A` повреждены, оно завершается неудачей.

При использовании AEAD можно оставить пустым `A` или `P`. Если связанные данные `A` пусты, AEAD становится обычным аутентифицированным шифром; если пуст `P`, это просто MAC.

> **Примечание.** На момент написания книги AEAD является современной нормой аутентифицированного шифрования. Поскольку почти все применяемые сегодня аутентифицированные шифры поддерживают связанные данные, далее в книге под аутентифицированными шифрами, если не указано иное, понимаются AEAD. Операции шифрования и расшифрования AEAD будут обозначаться соответственно AE и AD.

Предсказуемость и одноразовые номера

Напомним из главы 1, что безопасные схемы шифрования должны быть непредсказуемыми и при повторном шифровании одного открытого текста возвращать разные шифртексты, иначе атакующий сможет определить, был ли один открытый текст зашифрован дважды. Для непредсказуемости блочные и потоковые шифры принимают дополнительный параметр: начальное значение (IV) или одноразовый номер - число, которое разрешено использовать только один раз. Аутентифицированные шифры применяют тот же прием. Поэтому аутентифицированное шифрование записывается как `AE(K,P,A,N)`, где `N` - одноразовый номер. Операция шифрования обязана выбрать одноразовый номер, который еще никогда не использовался с тем же ключом.

Как и при блочном или потоковом шифровании, для правильного аутентифицированного расшифрования требуется одноразовый номер, использованный при шифровании. Поэтому расшифрование можно записать как `AD(K,C,A,T,N)=(P,A)`, где `N` - одноразовый номер, использованный для создания `C` и `T`.

Критерии хорошего аутентифицированного шифра

С начала 2000-х годов исследователи пытаются определить, каким должен быть хороший аутентифицированный шифр, но однозначного ответа до сих пор нет. Из-за множества входных данных AEAD, играющих разные роли, понятие безопасности определить труднее, чем для обычных шифров, которые лишь шифруют сообщение. Например, в исследовательской статье «Nonces Are Noticed: AEAD Revisited» предложена теоретическая основа шифрования с одноразовыми номерами. Тем не менее в этом разделе я обобщу важнейшие критерии оценки безопасности, производительности и функциональности аутентифицированного шифра.

Безопасность

Важнейшие критерии стойкости аутентифицированного шифра - его способность защищать конфиденциальность данных, то есть секретность открытого текста, а также подлинность и целостность связи, подобно способности MAC обнаруживать поврежденные сообщения. Аутентифицированный шифр должен выступать в обеих лигах: его конфиденциальность должна быть столь же сильной, как у лучших шифров, а подлинность - как у лучших MAC. Иными словами, если удалить из AEAD часть аутентификации, должен получиться безопасный шифр, а если удалить часть шифрования - сильный MAC.

Другой показатель безопасности аутентифицированного шифра - его хрупкость при повторении одноразовых номеров. Например, если одноразовый номер использован повторно, сможет ли атакующий расшифровать шифртексты или узнать различие между открытыми текстами?

Исследователи называют это свойство надежности устойчивостью к неправильному применению и разработали устойчивые к неправильному применению аутентифицированные шифры, чтобы оценивать последствия повторного одноразового номера и определять, будет ли при такой атаке нарушена конфиденциальность, подлинность или обе, а также какие сведения о зашифрованных данных, вероятнее всего, утекут.

Производительность

Как и для любого криптографического алгоритма, пропускная способность аутентифицированного шифра измеряется числом битов, обрабатываемых в секунду. Скорость зависит от числа операций алгоритма шифра и дополнительных затрат на аутентификацию. Дополнительные свойства безопасности аутентифицированных шифров снижают производительность. Однако оценка производительности шифра не сводится к чистой скорости. Важны также распараллеливаемость, структура и возможность потоковой обработки. Рассмотрим эти понятия подробнее.

Распараллеливаемость шифра характеризует его способность обрабатывать несколько блоков данных одновременно, не ожидая завершения обработки предыдущего блока.

Конструкции на основе блочных шифров легко распараллеливаются, если каждый блок можно обрабатывать независимо от остальных. Например, режим блочного шифра CTR из главы 4

распараллеливается, а режим шифрования CBC - нет, поскольку блоки сцеплены.

Внутренняя структура аутентифицированного шифра - еще один важный критерий производительности. Существует два основных типа структуры: однослойная и двухслойная. В двухслойной структуре, например в широко применяемом AES-GCM, один алгоритм обрабатывает открытый текст, а второй - полученный результат. Обычно первый слой является слоем шифрования, а второй - слоем аутентификации. Но, как можно ожидать, двухслойная структура усложняет реализацию и обычно замедляет вычисления.

Аутентифицированный шифр поддерживает потоковую, или оперативную, обработку, если способен обрабатывать сообщение блок за блоком и отбрасывать уже обработанные блоки. В отличие от него, непотоковый шифр должен хранить сообщение целиком, обычно потому, что требуется выполнить два последовательных прохода по данным: один от начала до конца, а второй - от конца к началу данных, полученных первым проходом.

Из-за потенциально высоких требований к памяти некоторые приложения не могут работать с непотоковыми шифрами. Например, маршрутизатор может получить зашифрованный блок данных, расшифровать его и вернуть блок открытого текста, прежде чем перейти к расшифрованию следующего блока сообщения, хотя получателю расшифрованного сообщения все равно придется проверить тег аутентификации, отправленный в конце потока расшифрованных данных.

Другие свойства

Функциональные критерии - это свойства шифра или его реализации, не связанные напрямую с безопасностью или производительностью. Например, одни аутентифицированные шифры разрешают связанным данным находиться только перед шифруемыми данными, поскольку для начала шифрования им нужен доступ к связанным данным. Другие требуют, чтобы связанные данные следовали после шифруемых, либо поддерживают включение связанных данных в любое место, даже между фрагментами открытого текста. Последний вариант лучше всего, поскольку позволяет защищать данные в любой возможной ситуации, но его и труднее всего безопасно спроектировать: дополнительные возможности часто означают дополнительную сложность и потенциальные уязвимости.

Еще один функциональный критерий - возможность использовать один основной алгоритм и для шифрования, и для расшифрования. Например, многие аутентифицированные шифры основаны на блочном шифре AES, который задает два похожих алгоритма для шифрования и расшифрования блока. Как обсуждалось в главе 4, режим CBC требует обоих алгоритмов, а режим CTR - только алгоритма шифрования. Точно так же аутентифицированным шифрам могут быть не нужны оба алгоритма. Дополнительная стоимость реализации и шифрования, и расшифрования несущественна для большинства программ, но часто заметна в недорогой специализированной аппаратуре, где стоимость реализации измеряют логическими вентилями или площадью кристалла, занятой криптографией.

Стандарт аутентифицированного шифра AES-GCM

AES-GCM - самый широко применяемый аутентифицированный шифр. Он основан на алгоритме AES, а режим работы счетчика Галуа (Galois counter mode, GCM) по существу представляет собой модификацию CTR с небольшим эффективным компонентом для вычисления тега аутентификации. На момент написания книги AES-GCM является стандартом NIST SP 800-38D, входит в Suite B АНБ и признан IETF для протоколов защищенной сети IPsec, SSH и TLS 1.2 и 1.3.

Примечание. Хотя GCM работает с любым блочным шифром, почти наверняка вы встретите его только с AES.

Внутреннее устройство GCM

На рисунке 8-2 показана работа AES-GCM: экземпляры AES, параметризованные секретным ключом `K`, преобразуют блок, состоящий из одноразового номера `N`, сцепленного со счетчиком, который начинается здесь с 1, а затем увеличивается до 2, 3 и так далее. После этого результат складывается посредством XOR с блоком открытого текста и дает блок шифртекста. Пока ничего нового по сравнению с режимом CTR.

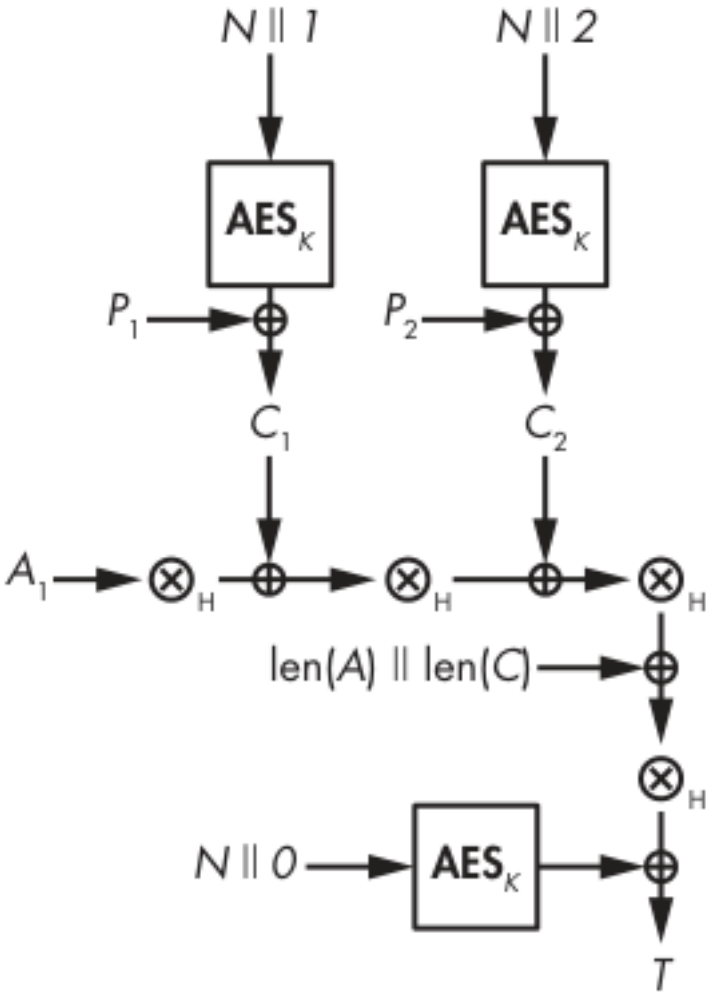

Рисунок 8-2. Режим AES-GCM, примененный к одному блоку связанных данных `A_1` и двум блокам открытого текста `P_1` и `P_2`. Обведенный знак умножения означает умножение многочлена на `H` - ключ аутентификации, полученный из `K`

Затем блоки шифртекста смешиваются сочетанием XOR и умножений, как будет показано далее. AES-GCM можно рассматривать как выполнение 1) шифрования в режиме CTR и 2) MAC над блоками шифртекста. Поэтому AES-GCM по существу представляет собой конструкцию «шифрование, затем MAC», где AES-CTR шифрует со 128-битным ключом `K` и 96-битным одноразовым номером `N`.

Для аутентификации шифртекста GCM использует MAC Вегмана - Картера, см. главу 7, который складывает посредством XOR значение AES(K,N || 0) с выходом универсальной хеш-функции GHASH. На рисунке 8-2 GHASH соответствует последовательности операций «x H», за которыми следует XOR с len(A) || len(C), то есть битовой длиной A, связанных данных, сцепленной с битовой длиной C, шифртекста.

Таким образом, значение тега аутентификации выражается как

```
T=GHASH(H,A,C) xor AES(K,N || 0),
```

где H - ключ хеширования, или ключ аутентификации. Он определяется как H=AES(K,0), то есть шифрование блока, равного последовательности нулевых байтов. Для ясности этот шаг не показан на рисунке 8-2.

> **Примечание.** В GCM функция GHASH не использует K напрямую, чтобы при компрометации ключа GHASH мастер-ключ K оставался секретным. Зная K, можно получить H, вычислив AES(K,0), но восстановить K из этого значения невозможно, поскольку здесь K служит ключом AES.

GHASH использует многочленную запись для умножения каждого блока шифртекста на ключ аутентификации H. Благодаря такому умножению GHASH быстро работает в аппаратуре и программах с помощью специальной инструкции умножения многочленов, имеющейся во многих распространенных микропроцессорах: CLMUL, carry-less multiplication, умножение без переносов.

Увы, GHASH далека от идеала. Во-первых, ее скорость неоптимальна. Даже с инструкцией CLMUL слой AES-CTR, шифрующий открытый текст, остается быстрее MAC GHASH. Во-вторых, GHASH трудно правильно реализовать. Даже опытные разработчики OpenSSL - безусловно, самого широко применяемого криптографического программного обеспечения в мире - допустили ошибку в GHASH AES-GCM. В одном коммите функция gcm_ghash_clmul содержала ошибку, позволявшую атакующим подделывать действительные MAC для AES-GCM. К счастью, инженеры Intel заметили ошибку до того, как она попала в следующий выпуск OpenSSL.

> **УМНОЖЕНИЕ МНОГОЧЛЕНОВ**
>
> Хотя оно сложнее классической целочисленной арифметики, для компьютеров умножение многочленов проще, поскольку в нем нет переносов. В частности, можно вычислять модульное умножение многочленов в структуре, определенной заданным многочленом; такие структуры называются фактор-кольцами. Предположим, требуется вычислить произведение многочленов (1+X+X^2) и (1+X^2) по модулю (1+X+X^3). Сначала два многочлена перемножаются как обычно, что дает следующее выражение. Два члена X^2 взаимно уничтожаются, поскольку мы работаем с булевыми многочленами, в мире которых 1+1=0:

```
(1+X+X^2) x (1+X^2)=1+X+X^3+X^4.
```

> Выполняется приведение по модулю: 1+X+X^3+X^4 по модулю 1+X+X^3 дает X+X^2, поскольку

```
1+X+X^3+X^4=(1+X) x (1+X+X^3)+X+X^2.
```

> В общем виде A+BC по модулю B по определению равно A.

Безопасность GCM

Главная слабость AES-GCM - хрупкость при повторении одноразового номера. Если два разных сообщения зашифровать с одним одноразовым номером N, атакующий, наблюдающий два шифртекста, сможет определить XOR соответствующих открытых текстов. Кроме того, он сможет восстановить ключ аутентификации H и использовать его для подделки тегов любого шифртекста, связанных данных или их комбинации.

Эта хрупкость становится понятнее при рассмотрении базовой алгебры вычислений AES-GCM, показанных на рисунке 8-2. Тег T вычисляется как `T=GHASH(H,A,C) xor AES(K,N || 0)`, где GHASH - универсальная хеш-функция с линейно связанными входами и выходами.

Если вычислить два тега T_1 и T_2 с одним одноразовым номером N, часть AES исчезнет. Пусть

```
T_1 =GHASH(H,A_1,C_1) xor AES(K,N || 0),
T_2 =GHASH(H,A_2,C_2) xor AES(K,N || 0).
```

Их XOR равен

```
T_1 xor T_2=() (GHASH(H,A_1,C_1) xor AES(K,N || 0))
xor (GHASH(H,A_2,C_2) xor AES(K,N || 0))
=() GHASH(H,A_1,C_1) xor GHASH(H,A_2,C_2).
```

Таким образом, при двукратном использовании одного одноразового номера атакующий может восстановить `GHASH(H,A_1,C_1) xor GHASH(H,A_2,C_2)` для известных A_1, C_1, A_2 и C_2. Линейность GHASH затем позволяет восстановить H.

В 2016 году исследователи просканировали интернет в поисках экземпляров AES-GCM, доступных через HTTPS-серверы, чтобы найти системы с повторяющимися одноразовыми номерами; см. исследовательскую статью. Они обнаружили 184 сервера с повторяющимися номерами, в том числе 23 сервера, всегда использовавших нулевую строку в качестве одноразового номера.

Эффективность GCM

Преимущество режима GCM в том, что при шифровании и расшифровании блоки обрабатываются независимо, что позволяет распараллелить вычисления. Однако вычисление GMAC не распараллеливается, поскольку после обработки GHASH всех связанных данных оно должно выполняться от начала до конца шифртекста. Из-за отсутствия распараллеливания любая система, которая сначала получает открытый текст, а затем связанные данные, должна дождаться чтения и хеширования всех связанных данных, прежде чем хешировать первый блок шифртекста.

Однако GCM поддерживает потоковую обработку: поскольку вычисления двух его слоев можно организовать конвейером, хранить все блоки шифртекста перед вычислением GHASH не требуется - GHASH обрабатывает каждый блок по мере его шифрования. Иными словами, P_1 шифруется в C_1, затем GHASH обрабатывает C_1, пока P_2 шифруется в C_2, после чего P_1 и C_1 больше не нужны, и так далее.

Режим аутентифицированного шифра OCB

Offset codebook (OCB), впервые разработанный в 2001 году, появился раньше GCM и, как GCM, создает аутентифицированный шифр из блочного, но делает это быстрее и проще. OCB пока не получил более широкого распространения, поскольку до 2021 года его использование было запатентовано и требовало лицензии изобретателя. Теперь любой может свободно применять OCB в любых приложениях.

В отличие от GCM, OCB объединяет шифрование и аутентификацию в одном слое обработки с одним ключом. Отдельного слоя аутентификации нет, поэтому OCB обеспечивает аутентификацию почти бесплатно и выполняет почти столько же вызовов блочного шифра, сколько выполнял бы неаутентифицированный шифр. OCB почти так же прост, как режим ECB, см. главу 4, но намного безопаснее.

Внутреннее устройство OCB

На рисунке 8-3 показана работа OCB: каждый блок открытого текста `P` шифруется в блок шифртекста `C=E(K,P xor O) xor O`, где `E` - функция шифрования блочного шифра. Здесь `O`, смещение, - значение, зависящее от ключа и одноразового номера и изменяемое для каждого нового обрабатываемого блока.

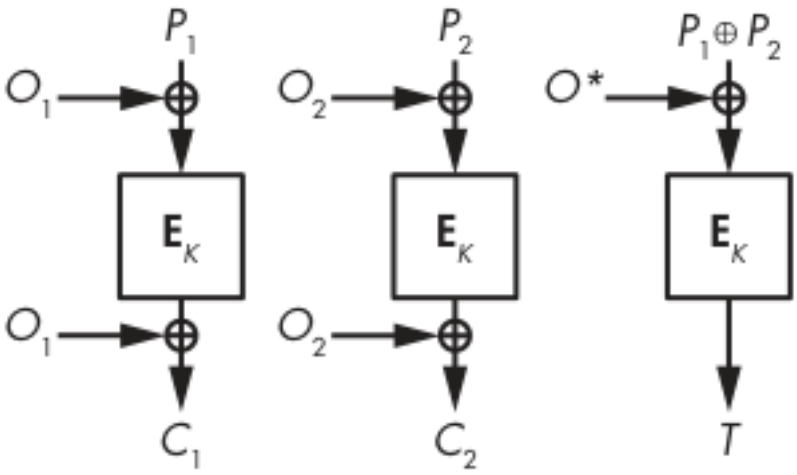

Рисунок 8-3. Процесс шифрования OCB для двух блоков открытого текста без связанных данных

Чтобы сформировать тег аутентификации, OCB сначала складывает блоки открытого текста посредством XOR и вычисляет `S=P_1 xor P_2 xor P_3 xor ...`. Затем тег вычисляется как `T=E(K,S xor O^*)`, где `O^*` - значение смещения, вычисленное из смещения последнего обработанного блока открытого текста.

Подобно AES-GCM, OCB поддерживает связанные данные в виде последовательности блоков `A_1`, `A_2` и так далее. Если зашифрованное OCB сообщение содержит связанные данные, тег аутентификации вычисляется по формуле

```
T=E(K,S xor O^*) xor E(K,A_1 xor O_1) xor E(K,A_2 xor O_2) xor ...,
```

где OCB определяет смещения, отличные от используемых для шифрования `P`.

В отличие от GCM и «шифрования, затем MAC», где для формирования тега объединяются блоки шифртекста, OCB вычисляет тег аутентификации, объединяя данные открытого текста. В этом нет ничего плохого, а безопасность OCB подтверждается строгими доказательствами.

> **Примечание.** Подробнее о реализации OCB см. RFC 7253 или статью Теда Кровеца и Филлипа Рогавея 2011 года «The Software Performance of Authenticated-Encryption Modes», посвященную новейшей и лучшей версии OCB - OCB3. Дополнительные сведения об OCB см. в FAQ.

Безопасность OCB

OCB немного менее хрупок при повторении одноразовых номеров, чем GCM. Если использовать одноразовый номер дважды, атакующий, видящий два шифртекста, может заметить, например, что третий блок открытого текста первого сообщения совпадает с третьим блоком второго. При использовании GCM атакующие способны обнаруживать не только совпадения, но и XOR-разности блоков в одинаковых позициях. Поэтому последствия повторения одноразовых номеров для GCM хуже, чем для OCB.

Как и в GCM, повторение одноразовых номеров ставит под угрозу подлинность OCB, но в меньшей степени. Например, атакующий может объединить блоки двух сообщений, аутентифицированных OCB, и создать другое зашифрованное сообщение с той же контрольной суммой и тегом, что у одного из исходных сообщений, но не сможет восстановить секретный ключ, как в GCM.

В 2023 году криптографы обнаружили, что применение OCB3 с очень короткими одноразовыми номерами длиной 6 битов значительно снижает его безопасность; см. статью. Обратите внимание, что версия OCB2 была взломана, как описано в этой статье.

Эффективность OCB

OCB и GCM примерно одинаково эффективны. Подобно GCM, OCB распараллеливается и поддерживает потоковую обработку. По чистой эффективности GCM и OCB выполняют примерно одинаковое число вызовов лежащего в основе блочного шифра, обычно AES, но OCB немного быстрее GCM, поскольку просто складывает открытый текст посредством XOR вместо сравнительно дорогостоящего вычисления вроде GHASH. В предыдущих поколениях микропроцессоров Intel AES-GCM был более чем в три раза медленнее AES-OCB, потому что инструкции AES и GHASH конкурировали за ресурсы CPU и не могли выполняться параллельно.

Важное различие реализаций OCB и GCM состоит в том, что для шифрования и расшифрования OCB нужны обе функции блочного шифра, что повышает стоимость аппаратных реализаций при ограниченной площади кристалла для криптографических компонентов. GCM, напротив, использует для шифрования и расшифрования только функцию шифрования.

Режим аутентифицированного шифра SIV

Synthetic IV (SIV) - режим аутентифицированного шифра, обычно используемый с AES. В отличие от GCM и OCB, SIV безопасен даже при двукратном использовании одного одноразового номера: атакующий, получивший два шифртекста, зашифрованных с одинаковым номером, сможет узнать лишь то, был ли один открытый текст зашифрован дважды. В отличие от GCM или OCB, он не сможет определить, совпадает ли первый блок двух сообщений, поскольку одноразовый номер, используемый для шифрования, сначала вычисляется как комбинация заданного номера и открытого текста.

Спецификация конструкции SIV более общая, чем GCM. Вместо подробного описания внутреннего устройства, как GHASH в GCM, SIV лишь определяет способ объединения шифра `E` и псевдослучайной функции PRF в аутентифицированный шифр: сначала вычисляется тег `T=PRF(K_1,N || P)`, а затем шифртекст `C=E(K_2,T,P)`, где `T` служит одноразовым номером `E`. Таким образом, SIV требуются два ключа `K_1` и `K_2` и одноразовый номер `N`.

Главное ограничение SIV - отсутствие потоковой обработки: после вычисления `T` весь открытый текст `P` должен оставаться в памяти. Иными словами, чтобы зашифровать с SIV открытый текст

объемом 100 Гбайт, сначала нужно сохранить все 100 Гбайт, чтобы SIV мог прочитать их при шифровании.

Документ RFC 5297, основанный на статье Филлипа Рогавея и Томаса Шримптона 2006 года «Deterministic Authenticated-Encryption», определяет SIV с CMAC-AES, конструкцией MAC на основе AES, в роли PRF и AES-CTR в роли шифра. В 2015 году была предложена более эффективная версия SIV - GCM-SIV, объединяющая быструю функцию GHASH из GCM с режимом SIV и почти не уступающая GCM в скорости. Однако, как и исходный SIV, GCM-SIV не поддерживает потоковую обработку. Подробнее см. https://eprint.iacr.org/2015/102.

AEAD на основе перестановки

Теперь рассмотрим совершенно другой подход к построению аутентифицированного шифра: вместо режима работы вокруг блочного шифра вроде AES возьмем шифр, строящий режим вокруг перестановки. Перестановка просто и обратимо преобразует входные данные в выходные того же размера без ключа - трудно представить более простой компонент. Еще лучше то, что получившийся AEAD быстр, имеет доказанную безопасность и устойчивее к повторному использованию одноразовых номеров, чем GCM и OCB.

На рисунке 8-4 показана работа AEAD на основе перестановки: начиная с фиксированного начального состояния `H_0`, вы складываете с внутренним состоянием посредством XOR ключ `K`, а затем одноразовый номер `N`, получая новое внутреннее состояние того же размера, что исходное. Затем новое состояние преобразуется перестановкой `P`, дающей следующее значение состояния. Теперь первый блок открытого текста `P_1` складывается посредством XOR с состоянием, а получившееся значение принимается как первый блок шифртекста `C_1`; `P_1` и `C_1` имеют один размер, но короче состояния.

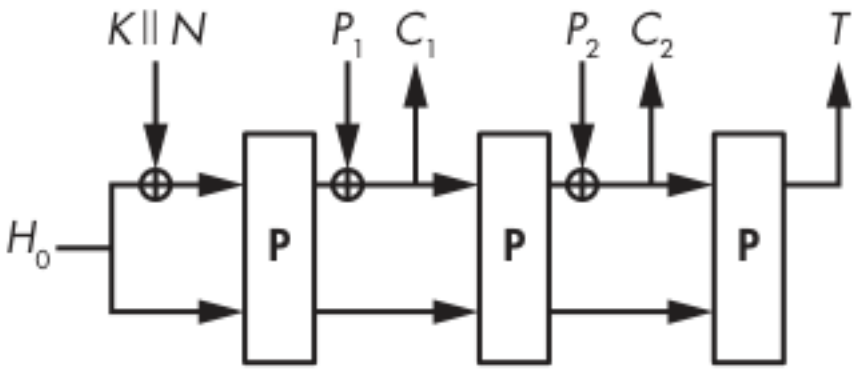

Рисунок 8-4. Аутентифицированный шифр на основе перестановки

Для шифрования второго блока состояние преобразуется посредством `P`, следующий блок открытого текста `P_2` складывается посредством XOR с текущим состоянием, а получившееся значение принимается как `C_2`. Затем процесс повторяется для всех блоков открытого текста, а после последнего вызова `P` биты внутреннего состояния берутся как тег аутентификации `T`, что показано в правой части рисунка 8-4.

> **Примечание.** Режим на рисунке 8-4 можно адаптировать для поддержки связанных данных, но процесс немного сложнее, поэтому я опущу его описание.

Безопасная конструкция аутентифицированного шифра на основе перестановки должна удовлетворять нескольким требованиям. Во-первых, входные значения следует складывать посредством XOR только с частью состояния: чем больше эта часть, тем больше контроля над внутренним состоянием получает успешный атакующий и тем ниже безопасность шифра. Вся безопасность опирается на секретность внутреннего состояния.

Кроме того, блоки нужно правильно дополнять дополнительными битами так, чтобы любые два разных сообщения давали разные результаты. В качестве контрпримера: если последний блок открытого текста короче полного, нельзя просто дополнять его нулями. Иначе блок открытого текста, например, из 2 байтов `0000` превратится в полный блок `0000...0000`, как и блок из 3 байтов `000000`. В результате оба сообщения получат один тег, хотя различаются размером.

Если повторно использовать одноразовый номер в таком шифре на основе перестановки, последствия не столь тяжелы, как в GCM или OCB: стойкость тега аутентификации не нарушается. При повторении одноразового номера атакующий узнает только то, начинаются ли два зашифрованных сообщения с одного значения, и длину этого общего значения, или префикса. Например, при шифровании двух шестиблочных сообщений `ABCXYZ` и `ABCDYZ`, где каждая буква обозначает блок, с одним одноразовым номером могут получиться два шифртекста `JKLTUV` и `JKLMNO` с одинаковыми префиксами. Но атакующий не сможет узнать, что у двух открытых текстов совпадали последние два блока `YZ`.

С точки зрения производительности шифры на основе перестановок обладают преимуществами одного слоя операций, потоковой обработки и одного основного алгоритма для шифрования и расшифрования. Однако они не распараллеливаются подобно GCM или OCB, поскольку каждый новый вызов `P` должен ждать завершения предыдущего.

> **Примечание.** Если вам захочется взять любимую перестановку и изобрести собственный аутентифицированный шифр, пожалуйста, не делайте этого. Вы, скорее всего, ошибетесь в деталях и получите небезопасный шифр. Прочитайте спецификации, написанные опытными криптографами, для таких алгоритмов, как Keyak, производный от Keccak, конструкция Duplex и deck-функции на https://keccak.team. Вы увидите, что шифры на основе перестановок сложнее, чем кажутся на первый взгляд.

Что может пойти не так

Поверхность атаки у аутентифицированных шифров больше, чем у хеш-функций или блочных шифров, поскольку они стремятся обеспечить одновременно конфиденциальность и подлинность. Они принимают несколько разных входных значений и должны оставаться безопасными независимо от того, содержат ли входные данные только связанные данные без шифруемых, открытые тексты огромного размера или ключи разной длины. Они также должны быть безопасны для всех значений одноразового номера против атакующих, собирающих множество пар «сообщение/тег», и в некоторой степени против случайного повторения одноразовых номеров.

Требований много, и даже у AES-GCM есть несколько недостатков.

AES-GCM и слабые ключи хеширования

Одна из слабостей AES-GCM находится в алгоритме аутентификации GHASH: некоторые значения ключа хеширования `H` значительно упрощают атаки на механизм аутентификации GCM. В частности, если `H` принадлежит некоторым строго определенным математическим подгруппам всех 128-битных строк, атакующие могут угадать действительный тег аутентификации некоторого сообщения, просто переставляя блоки предыдущего сообщения.

Чтобы понять эту слабость, рассмотрим работу GHASH.

Внутреннее устройство GHASH

Как показано на рисунке 8-2, GHASH начинает со 128-битного значения `H`, первоначально равного `AES(K,0)`, а затем многократно вычисляет

```
X_i=(X_(i-1) xor C_i) x H,
```

начиная с `X_0=0` и обрабатывая блоки шифртекста `C_1`, `C_2` и так далее. GHASH возвращает итоговое `X_i` для вычисления окончательного тега.

Для простоты предположим, что все значения `C_i` равны 1, так что для любого `i`

```
C_i x H=1 x H=H.
```

Теперь вычисление `X_1` по первому уравнению дает

```
X_1=(X_0 xor C_1) x H=(0 xor 1) x H=H,
```

где `X_0` заменено на 0, а `C_1` - на 1, так что

```
(0 xor 1)=1.
```

Благодаря дистрибутивности `x` относительно `xor` подставим вместо `X_1` значение `H`, вместо `C_2` - 1 и вычислим следующее значение `X_2`:

```
X_2=(X_1 xor C_2) x H=(H xor 1) x H=H^2 xor H,
```

где `H^2` - квадрат `H`, то есть `H x H`.

Теперь выведем `X_3`, подставив выражение для `X_2`:

```
X_3=(X_2 xor C_3) x H=(H^2 xor H xor 1) x H=H^3 xor H^2 xor H.
```

Далее получаем `X_4=H^4 xor H^3 xor H^2 xor H` и так далее, а итоговое `X_i` в конце равно

```
X_n=H^n xor H^(n-1) xor H^(n-2) xor ... xor H^2 xor H.
```

Напомним, что все блоки `C_i` были положены равными 1. Если же они произвольны, получится

```
X_n=(C_1 x H^n) xor (C_2 x H^(n-1)) xor (C_3 x H^(n-2)) xor ... xor (C_(n-1) x H^2) xor (
    C_n x H).
```

Затем GHASH складывает с последним `X_n` посредством XOR длину сообщения, умножает результат на `H` и складывает полученное значение посредством XOR с `AES(K,N || 0)`, создавая окончательный тег аутентификации `T`.

Где все ломается

Что теперь может пойти не так? Начнем с простейших случаев:

• Если H=0, то X_n=0 независимо от значений C_i, а значит, и от сообщения. Иными словами, при H=0 у всех сообщений будет один тег аутентификации.
• Если H=1, тег представляет собой просто XOR блоков шифртекста, и их перестановка даст тот же тег аутентификации.

Поскольку 0 и 1 - лишь два из 2^128 возможных значений H, вероятность их появления равна всего 2/2^128=1/2^127. Но существуют и другие слабые значения: все значения H, принадлежащие короткому циклу при возведении в степень i. Например, значение H 10d04d25f93556e69f58ce2f8d035a4 принадлежит циклу длиной 5, поскольку удовлетворяет H^5=H, а следовательно, H^e=H для любого e, кратного 5. Именно так определяется цикл относительно пятых степеней. Поэтому в приведенном выше выражении окончательного значения GHASH X_n перестановка блока C_n, умножаемого на H, и блока C_(n-4), умножаемого на H^5, не изменяет тег аутентификации, что равносильно подделке. Атакующий может использовать это свойство для создания нового сообщения и действительного тега к нему без знания ключа, что для безопасного аутентифицированного шифра должно быть невозможно.

Предыдущий пример основан на цикле длиной 5, но существует множество циклов большей длины и, следовательно, множество значений H, более слабых, чем должны быть. В итоге в маловероятном случае, когда H принадлежит короткому циклу значений, атакующие могут подделывать сколько угодно тегов аутентификации. Однако, не зная H или K, они не могут определить длину цикла H. Хотя атакующие не способны использовать эту уязвимость, ее можно избежать тщательным выбором многочлена для приведения по модулю.

> **Примечание.** Подробнее об этой атаке см. статью Маркку-Юхани О. Сааринена «Cycling Attacks on GCM, GHASH and Other Polynomial MACs and Hashes».

AES-GCM и короткие теги

На практике AES-GCM обычно возвращает 128-битные теги, но способен формировать теги любой длины. К сожалению, при использовании более коротких тегов вероятность подделки значительно возрастает.

При 128-битном теге атакующий, пытающийся выполнить подделку, должен добиться успеха с вероятностью 1/2^128, поскольку существует 2^128 возможных 128-битных тегов. В общем случае для n-битного тега вероятность успеха должна быть равна 1/2^n, где 2^n - число возможных значений n-битного тега. Но при более коротких тегах из-за выходящих за рамки этого обсуждения слабостей структуры GCM вероятность подделки намного выше 1/2^n. Например, при 32-битном теге атакующий, знающий тег аутентификации некоторого сообщения размером 2 Мбайт, добьется успеха с вероятностью 1/2^16 вместо 1/2^32.

В общем случае для n-битных тегов вероятность подделки равна не 1/2^n, а 2^m/2^n, где 2^m - число блоков самого длинного сообщения, тег которого наблюдал успешный атакующий. Например, при 48-битных тегах и сообщениях объемом 4 Гбайт, то есть 2^28 блоков по 16 байтов, вероятность подделки равна 2^28/2^48=1/2^20, или примерно одному шансу на миллион. Для криптографии это сравнительно высокая вероятность. Подробнее об атаке см. статью Нильса Фергюсона 2005 года «Authentication Weaknesses in GCM».

Дополнительная литература

Чтобы больше узнать об аутентифицированных шифрах, посетите главную страницу CAESAR - Competition for Authenticated Encryption: Security, Applicability, and Robustness по адресу http://competitions.cr.yp.to/caesar.html. Конкурс CAESAR начался в 2012 году по образцу конкурсов AES и SHA-3, хотя организован не NIST.

Конкурс CAESAR привлек впечатляющее число инновационных конструкций: от режимов, похожих на OCB, до режимов на основе перестановок, таких как NORX и Keyak, а также совершенно оригинальных алгоритмов, таких как AEZ и AEGIS. В 2019 году CAESAR завершился выбором портфеля из семи алгоритмов, разделенных на три категории; они перечислены на официальной странице.

Другим конкурсом алгоритмов аутентифицированного шифрования был проект NIST Lightweight Cryptography. Он проходил с 2017 по 2023 год и был направлен на стандартизацию алгоритмов, оптимизированных для сред с ограниченными ресурсами - памятью, размером процессора и так далее, - например встраиваемых платформ. Победителем стало семейство алгоритмов на основе перестановок Ascon, подробно представленное на официальном сайте. Другие алгоритмы-кандидаты и работы, представленные на семинарах проекта, находятся на странице NIST.

В этой главе основное внимание уделялось GCM, но в реальных приложениях используется и несколько других режимов. В частности, режимы counter with CBC-MAC (CCM) и EAX конкурировали с GCM за стандартизацию в начале 2000-х годов. Хотя был выбран GCM, его конкуренты применяются в некоторых приложениях. Например, CCM используется в протоколе шифрования Wi-Fi WPA2. Рекомендуется прочитать спецификации этих шифров и сравнить их безопасность и производительность.

На этом обсуждение криптографии с симметричным ключом завершено! Вы познакомились с блочными и потоковыми шифрами, хеш-функциями с ключом и без него, а теперь и с аутентифицированными шифрами - всеми основными компонентами криптографии, работающими с симметричным ключом или вовсе без ключа. Прежде чем перейти к асимметричной криптографии, в главе 9 мы рассмотрим информатику и математику, необходимые для понимания таких асимметричных схем, как RSA в главе 10 и Диффи - Хеллман в главе 11.

Глава 9. Трудные задачи

Трудные вычислительные задачи - краеугольный камень современной криптографии. Это задачи, для которых даже лучший алгоритм не найдет решения до того, как погаснет Солнце.

В 1970-х годах строгое исследование трудных задач породило новую область науки - теорию вычислительной сложности, которая оказала огромное влияние на криптографию и многие другие области, включая экономику, физику и биологию. В этой главе вы познакомитесь с концептуальными инструментами теории сложности, необходимыми для понимания основ криптографической безопасности. Я также представлю трудные задачи, лежащие в основе схем с открытым ключом, таких как шифрование RSA и согласование ключей Диффи - Хеллмана. Мы затронем глубокие понятия, но я сведу технические подробности к минимуму и лишь коснусь поверхности. Тем не менее я надеюсь, вы увидите красоту того, как криптография использует теорию вычислительной сложности для максимальной уверенности в безопасности.

Вычислительная трудность

Вычислительная задача - вопрос, на который можно ответить, выполнив достаточный объем вычислений, например: «Является ли 217 простым числом?» или «Сколько букв `i` в слове `incomprehensibilities`?» Первый вопрос является задачей разрешимости, поскольку на него можно ответить «да» или «нет», а второй - задачей поиска.

Вычислительная трудность - свойство вычислительных задач, для которых не существует алгоритма, выполняющегося за разумное время. Такие задачи также называются практически неразрешимыми. Вычислительная трудность не зависит от вида вычислительного устройства: центрального процессора общего назначения (CPU), графического процессора (GPU), интегральной схемы или механической машины Тьюринга. Одно из первых открытий теории вычислительной сложности состоит в эквивалентности всех моделей вычислений. Если одно вычислительное устройство способно эффективно решить задачу, любое другое также сможет сделать это после переноса алгоритма на его язык. Исключение составляют квантовые компьютеры, которые мы обсудим в главе 14. Поэтому при обсуждении вычислительной трудности не требуется указывать конкретное вычислительное устройство или аппаратуру: речь будет идти просто об алгоритмах.

Чтобы оценивать вычислительную трудность, сначала нужен способ измерить сложность алгоритма, то есть время его выполнения. Затем времена выполнения делятся на трудные и легкие.

Время выполнения

Вычислительная сложность алгоритма - приблизительное число выполняемых им операций как функция размера входных данных. Размер можно считать в битах или в числе входных элементов. Например, рассмотрим записанный псевдокодом алгоритм из листинга 9-1. Он ищет значение `x` в массиве из `n` элементов и возвращает его индекс либо `-1`, если `x` не найден.

```
search(x, array, n) {
    for i from 0 to n - 1 {
        if (array[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Листинг 9-1. Простой алгоритм поиска, сложность которого линейна относительно длины массива n

Этот алгоритм использует цикл `for` для поиска заданного значения `x` в массиве, перебирая значения переменной `i` начиная с 0. Он проверяет, равно ли значение в позиции `i` массива значению `x`. Если равно, возвращается позиция `i`. В противном случае `i` увеличивается и проверяется следующая позиция, пока алгоритм не достигнет `n-1`, последней позиции массива, после чего возвращается `-1`.

Для такого алгоритма сложность считается как число итераций цикла `for`: 1 в лучшем случае, если `x` равно `array[0]`; `n` в худшем, если `x` равно `array[n-1]` или отсутствует в `array`; и в среднем `n/2`, если `x` равномерно случайно распределен по одной из `n` ячеек массива. Для массива в 10 раз большего размера алгоритм будет в 10 раз медленнее. Поэтому сложность пропорциональна `n`, или «линейна» по `n`. Сложность, линейная относительно размера входных данных, считается быстрой, в отличие от экспоненциальной. Хотя обработка больших входных данных в этом примере медленнее, вычислительная стоимость не растет экспоненциально, а остается пропорциональной размеру таблицы.

Однако многие полезные алгоритмы медленнее и имеют сложность выше линейной. Классический пример - алгоритмы сортировки: чтобы упорядочить список из `n` значений в случайном порядке, в худшем случае требуется `n*log n` базовых операций. Иногда это называется линейно-логарифмической сложностью. Поскольку `n*log n` растет быстрее `n`, скорость сортировки с ростом `n` снижается быстрее, чем пропорционально. И все же такие алгоритмы сортировки остаются в области практических вычислений, которые можно выполнить за разумное время.

Обычно неразумны сложности, экспоненциально зависящие от размера входных данных. В какой-то момент предел осуществимого будет достигнут даже при сравнительно коротких входах. Рассмотрим простейший пример из криптоанализа - полный перебор секретного ключа. Напомним из главы 1, что для открытого текста `P` и шифртекста `C=E(K,P)` восстановление `n`-битного симметричного ключа требует не более `2^n` попыток, поскольку существует `2^n` возможных ключей. Это пример экспоненциально растущей сложности. Задачу с экспоненциальной сложностью практически невозможно решить, потому что с ростом `n` требуемые усилия быстро становятся неосуществимыми.

Можно возразить, что мы сравниваем несопоставимое: в функции `search()` из листинга 9-1 считалось число операций `if (array[i] == x)`, а при восстановлении ключа - число шифрований, каждое из которых в тысячи раз медленнее одного сравнения `==`. Эта кажущаяся непоследовательность может иметь значение при сравнении двух алгоритмов с очень близкими сложностями, но обычно несущественна, поскольку число операций влияет сильнее стоимости отдельной операции. Кроме того, при оценке сложности игнорируют постоянные множители: когда говорят, что алгоритм требует порядка `n^3` операций, то есть имеет кубическую сложность, на самом деле он может выполнять `41* n^3` или даже `12 345* n^3` операций. Но с ростом `n` постоянные множители теряют значение настолько, что ими можно пренебречь. Анализ сложности посвящен теоретической трудности как функции размера входных данных; точное число тактов CPU на вашем компьютере его не интересует.

Для выражения сложности часто применяется нотация `O()`, «большое O». Например, `O(n^3)` означает, что сложность растет не быстрее `n^3`, если не учитывать возможные постоянные множители. `O()` обозначает верхнюю границу сложности алгоритма. Запись `O(1)` означает, что

алгоритм выполняется за постоянное время, то есть его время работы не зависит от длины входа. Например, алгоритм, определяющий четность целого числа по его младшему значащему биту (least significant bit, LSB) и возвращающий «четное», если он равен нулю, и «нечетное» в противном случае, выполняет одну и ту же работу с одной стоимостью независимо от длины целого числа.

Чтобы увидеть разницу между линейной, квадратичной и экспоненциальной временной сложностью, посмотрим на рост `O(n)`, линейной, `O(n^2)`, квадратичной, и `O(2^n)`, экспоненциальной, на рисунке 9-1.

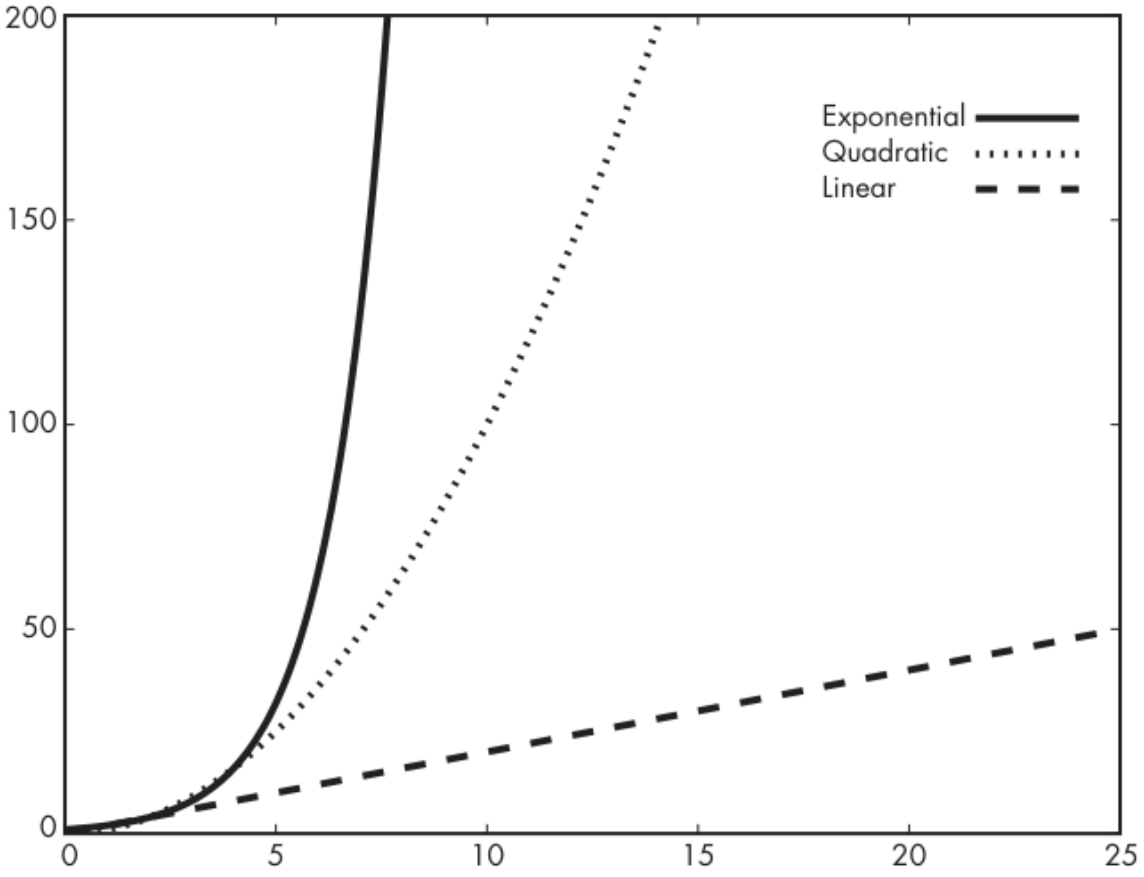

Рисунок 9-1. Рост экспоненциальной, квадратичной и линейной сложности - от самой быстрорастущей до самой медленнорастущей

Экспоненциальная сложность означает, что задачу практически невозможно решить, а линейная - что решение осуществимо; квадратичная сложность находится где-то между ними.

Полиномиальное и сверхполиномиальное время

Квадратичная сложность `O(n^2)`, средняя кривая на рисунке 9-1, - частный случай более широкого класса полиномиальных сложностей `O(n^k)`, где `k` - некоторое фиксированное число, например 3, 2,373, `7/10` или квадратный корень из 17. Алгоритмы полиномиального времени исключительно важны в теории сложности и криптографии, поскольку именно они определяют практическую осуществимость. Если алгоритм работает за полиномиальное время, сокращенно polytime, он завершается за приемлемое время даже при больших входных данных. Поэтому для теоретиков сложности и криптографов полиномиальное время является синонимом эффективности.

Напротив, алгоритмы сверхполиномиального времени, то есть `O(f(n))`, где `f(n)` растет быстрее любого многочлена, можно считать непрактичными. Я говорю именно о сверхполиномиальном, а не только об экспоненциальном времени, поскольку между полиномиальной и хорошо известной

экспоненциальной сложностью O(2^n) существуют промежуточные сложности, например O(n^(log n)), как показано на рисунке 9-2.

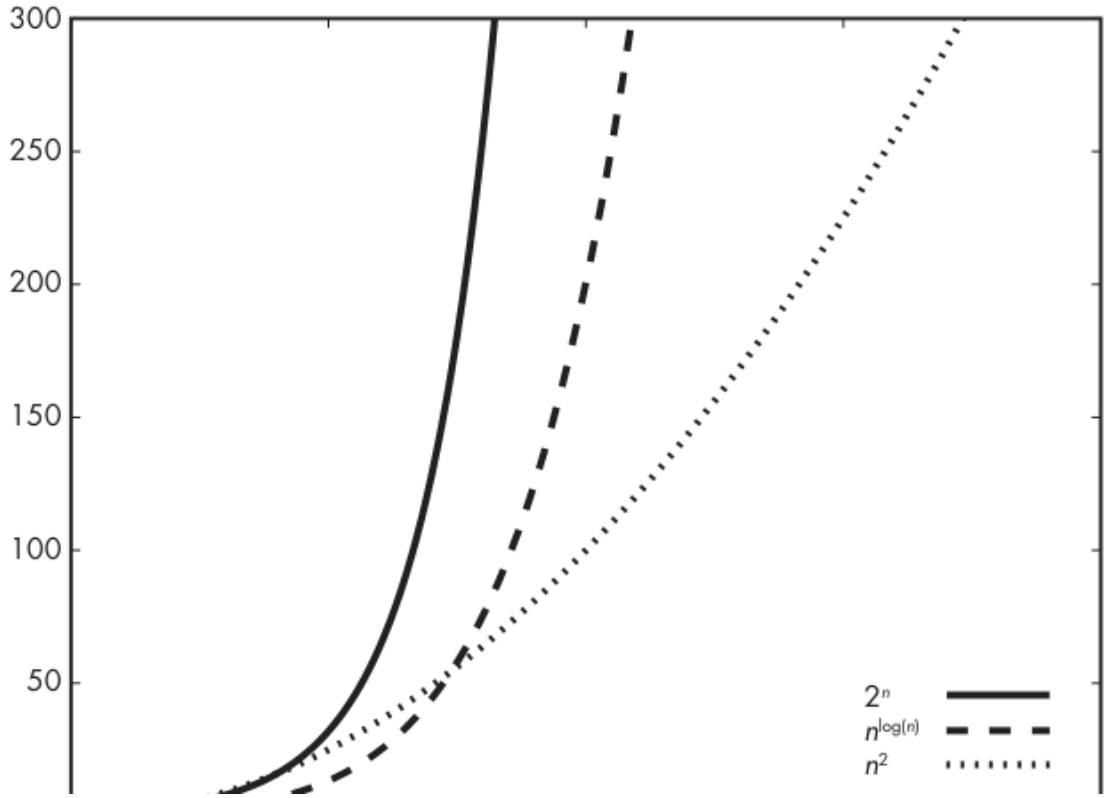

Рисунок 9-2. Рост функций 2^n, n^(log n) и n^2 - от самой быстрорастущей до самой медленнорастущей

O(n^2) или O(n^3) может быть эффективной, но O(n^(99 999 999 999)) очевидно не является таковой. Иными словами, полиномиальное время быстро на практике, пока показатель степени не слишком велик. К счастью, все найденные алгоритмы полиномиального времени для реальных задач имеют небольшие показатели. Например, O(n^(2,373)) - теоретическая временная сложность лучшего известного алгоритма умножения двух матриц n* n, хотя на практике он никогда не используется. Прорывной детерминированный алгоритм полиномиального времени 2002 года для определения простоты n-битных чисел первоначально имел сложность O(n^12), но позже был улучшен до O(n^6). Поэтому полиномиальное время, возможно, не идеально определяет практичность алгоритма, но это лучшее доступное определение.

Соответственно, задачу, которую нельзя решить алгоритмом полиномиального времени, можно считать непрактичной, или трудной. Например, при прямом переборе ключа нельзя превзойти сложность O(2^n), если только сам шифр каким-либо образом не взломан.

> **Примечание.** Экспоненциальная сложность O(2^n) - не худший вариант. Некоторые сложности растут еще быстрее и характеризуют еще более медленные алгоритмы: например, O(n^n) или экспоненциальный факториал O(n^(f(n-1))), где функция f рекурсивно определяется для любого x как f(x)=x^(f(x-1)). На практике алгоритмы с такими нелепыми сложностями вам никогда не встретятся.

Вы знаете, что сложность полного перебора ключа O(2^n) невозможно превзойти, пока шифр безопасен, но в общем случае самый быстрый способ решения вычислительной задачи известен не всегда. Значительная часть исследований в теории сложности посвящена доказательству границ сложности времени выполнения алгоритмов, решающих заданную задачу. Чтобы облегчить работу, теоретики сложности распределили вычислительные задачи по разным группам, или

классам, в зависимости от усилий, необходимых для решения.

Классы сложности

В математике класс - группа объектов с некоторым общим свойством. Например, все вычислительные задачи, решаемые за время `O(n^2)`, которое теоретики сложности обозначают просто `TIME(n^2)`, образуют один класс. Аналогично, `TIME(n^3)` - класс задач, решаемых за время `O(n^3)`, `TIME(2^n)` - класс задач, решаемых за время `O(2^n)`, и так далее. По той же причине, по которой суперкомпьютер способен вычислить все, что способен ноутбук, любая задача, решаемая за `O(n^2)`, решается и за `O(n^3)`. Поэтому любая задача класса `TIME(n^2)` принадлежит также классу `TIME(n^3)`, оба они принадлежат `TIME(n^4)` и так далее. Объединение всех классов `TIME(n^k)` для любых констант `k` называется P, что означает polynomial time, полиномиальное время.

Если вы программировали, то знаете, что кажущийся быстрым алгоритм все равно может обрушить систему, исчерпав всю память. При выборе алгоритма нужно учитывать не только временную сложность, но и объем используемой памяти, то есть пространственную сложность. Это особенно важно, поскольку один доступ к памяти обычно на несколько порядков медленнее базовой арифметической операции CPU.

Формально расход памяти алгоритма определяется как функция длины входа `n` точно так же, как временная сложность. Класс задач, решаемых с использованием `f(n)` битов памяти, называется `SPACE(f(n))`. Например, `SPACE(n^3)` - класс задач, решаемых с использованием порядка `n^3` битов памяти. Подобно тому как P является объединением всех `TIME(n^k)`, объединение всех задач `SPACE(n^k)` называется PSPACE.

Хотя чем меньше памяти, тем лучше, полиномиальный объем памяти не обязательно означает практичность алгоритма. Например, полный перебор ключа требует ничтожного объема памяти, но чертовски медленен. В общем случае алгоритм может выполняться вечно, даже используя всего несколько байтов памяти.

Любой задаче, решаемой за время `f(n)`, требуется не более `f(n)` памяти, поэтому `TIME(f(n))` входит в `SPACE(f(n))`. За время `f(n)` можно записать не более `f(n)` битов, поскольку предполагается, что запись или чтение одного бита занимает одну единицу времени; следовательно, любая задача из `TIME(f(n))` не может использовать больше `f(n)` памяти. Поэтому P является подмножеством PSPACE.

Недетерминированное полиномиальное время

NP, nondeterministic polynomial time, недетерминированное полиномиальное время, - второй по важности класс сложности после P, класса всех алгоритмов полиномиального времени.

NP - класс задач разрешимости, для которых решение можно проверить за полиномиальное время, то есть эффективно, хотя найти решение может быть трудно. Под проверкой я подразумеваю, что, имея потенциальное решение, можно выполнить некоторый алгоритм полиномиального времени и установить, действительно ли решение найдено. Например, задача определения того, существует ли ключ `K`, такой что `C=E(K,P)`, при заданных `P` и `C` для симметричной криптосистемы `E` принадлежит NP. Причина в том, что для ключа-кандидата `K_0` можно проверить его правильность, установив равенство `E(K_0,P)` и `C`. Найти потенциальный ключ, то есть решение, за полиномиальное время нельзя, если он существует, но проверить правильность ключа можно.

Теперь контрпример: что насчет атак с известным шифртекстом? На этот раз даны лишь некоторые значения `E(K,P)` для случайных неизвестных открытых текстов `P`. Если `P` неизвестны, проверить правильность потенциального ключа `K_0` невозможно. Иными словами, задача восстановления ключа при атаке с известным шифртекстом не принадлежит NP, а тем более P, поскольку ее нельзя выразить как задачу разрешимости.

Другой пример задачи, не принадлежащей NP, - проверка отсутствия решения. Проверка правильности решения сводится к выполнению некоторого алгоритма с решением-кандидатом в качестве входа и проверке возвращенного значения. Но для подтверждения отсутствия решения может потребоваться перебрать все возможные входы. Если их экспоненциально много, эффективно доказать отсутствие решения невозможно. Отсутствие решения трудно показать для самых сложных задач класса NP - так называемых NP-полных задач.

NP-полные задачи

NP-полные задачи - самые трудные задачи разрешимости класса NP; способы решения их худших случаев за полиномиальное время неизвестны. Когда в 1970-х годах теоретики сложности разработали теорию NP-полноты, они обнаружили, что самые трудные задачи NP в основе своей одинаково трудны. Это было доказано демонстрацией того, что любое эффективное решение любой NP-полной задачи можно превратить в эффективное решение любой другой NP-полной задачи. Иными словами, если вы умеете эффективно решать хотя бы одну NP-полную задачу, то сможете решить их все, а вместе с ними все задачи NP. Как это возможно?

NP-полные задачи выглядят по-разному, но с математической точки зрения принципиально похожи. Любую NP-полную задачу можно свести к любой другой так, что способность решить вторую подразумевает способность решить первую. Помните, что NP содержит задачи разрешимости, а не задачи поиска. Алгоритм решения задачи поиска можно эффективно преобразовать в алгоритм соответствующей задачи разрешимости, но обратное преобразование возможно не всегда. К счастью, для NP-полных задач оно возможно, чем и объясняется, почему эти два вида часто смешивают.

Вот несколько примеров NP-полных задач:

Задача коммивояжера. Даны множество точек на карте, например городов, расстояния между каждой парой точек и максимальное расстояние `x`. Нужно определить, существует ли путь, посещающий каждую точку, общая длина которого меньше `x`. Обратите внимание, что такой путь можно найти практически с той же сложностью, что решить задачу разрешимости, но определение оптимальности пути не принадлежит NP, поскольку правильность решения невозможно эффективно проверить.

Задача о клике. Даны число `x` и граф, то есть множество узлов, соединенных ребрами, как на рисунке 9-3. Нужно определить, существует ли множество не более чем из `x` узлов, каждый из которых соединен со всеми остальными.

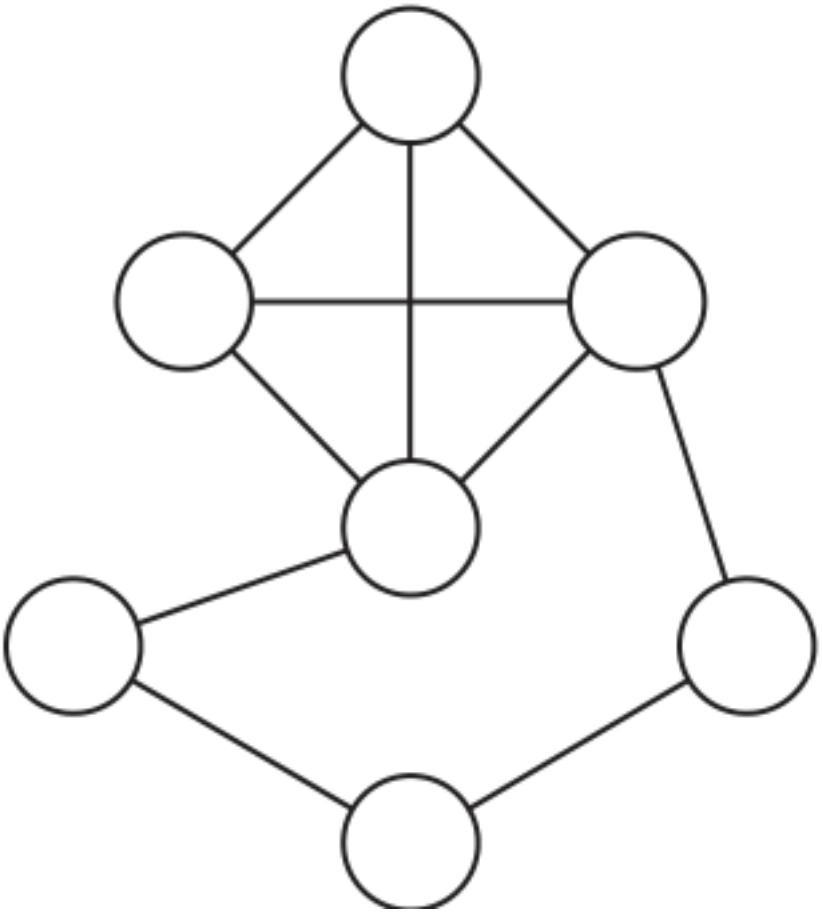

Рисунок 9-3. Граф, содержащий клику из четырех узлов. Общая задача поиска в графе клики заданного размера, то есть множества узлов, каждый из которых соединен со всеми остальными, является NP-полной

Задача о рюкзаке. Даны два числа x и y и набор предметов, для каждого из которых известны ценность и вес. Нужно определить, существует ли группа предметов с общей ценностью не менее x и общим весом не более y.

NP-полные задачи встречаются повсюду: от планирования, где задания с заданным приоритетом и длительностью нужно распределить по одному или нескольким процессорам, соблюдая приоритет и минимизируя общее время выполнения, до удовлетворения ограничений, где требуется определить значения, удовлетворяющие набору математических ограничений, например логических уравнений. Было доказано, что даже задача победы в некоторых видеоиграх NP-трудна, в том числе для известных игр Tetris, Super Mario Bros., Pokémon и Candy Crush Saga. Например, статья «Classic Nintendo Games are (Computationally) Hard» рассматривает «задачу разрешимости достижимости», чтобы определить возможность добраться до целевой точки из заданной начальной; см. https://arxiv.org/abs/1203.1895.

Некоторые из этих задач видеоигр не менее трудны, чем NP-полные, и называются NP-трудными. Задача, не обязательно разрешимости, является NP-трудной, если она по меньшей мере так же трудна, как NP-полные задачи разрешимости, и любой метод ее решения можно эффективно использовать для решения NP-полных задач.

NP-полные задачи обязаны быть задачами разрешимости, то есть иметь ответ «да» или «нет». Поэтому, строго говоря, задачи вычисления «лучшего» значения решения не могут быть NP-полными, но могут быть NP-трудными. Например, в задаче коммивояжера вопрос «Существует ли путь, посещающий все точки и имеющий длину меньше x?» является NP-полным, а требование

«Найти самый быстрый путь, посещающий все точки» - NP-трудным.

Обратите внимание, что не все экземпляры NP-трудных задач действительно трудно решать. Некоторые экземпляры могут эффективно решаться благодаря малому размеру или особой структуре. Возьмем, например, граф на рисунке 9-3. В нем легко заметить клику - четыре верхних соединенных узла. Хотя упомянутая задача поиска клики NP-трудна, здесь нет ничего трудного. NP-трудность означает не то, что трудны все экземпляры задачи, а то, что некоторые становятся трудными с ростом размера. Именно поэтому криптографов больше интересуют задачи, трудные в среднем, а не только в худшем случае.

Проблема P против NP

Если бы самые трудные задачи NP можно было решать за полиномиальное время, то и все задачи NP решались бы за полиномиальное время, следовательно, NP равнялось бы P. Такое равенство кажется абсурдным: разве не 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 потому эта задача не может принадлежать P?

Как бы безумно это ни звучало, никто не доказал различие P и NP, несмотря на премию в миллион долларов. Институт математики Клэя вручит ее тому, кто докажет либо `P!= NP`, либо `P=NP`. Эту проблему, известную как P против NP, знаменитый теоретик сложности Скотт Ааронсон назвал «одним из глубочайших вопросов, когда-либо заданных человечеством». Подумайте: если бы P равнялось NP, любое легко проверяемое решение было бы легко найти. Вся практическая криптография стала бы небезопасной, поскольку симметричные ключи можно было бы эффективно восстанавливать, а хеш-функции - обращать.

Но не паникуйте: большинство теоретиков сложности полагают, что P не равно NP и потому P является строгим подмножеством NP, как показано на рисунке 9-4, где NP-полные задачи образуют другое подмножество NP, не пересекающееся с P. Иными словами, задачи, выглядящие трудными, действительно трудны. Просто математически доказать это сложно. Для доказательства `P=NP` достаточно одного алгоритма полиномиального времени для NP-полной задачи, а доказать отсутствие такого алгоритма принципиально труднее. Это не мешает математикам предлагать простые доказательства, обычно очевидно ошибочные, но нередко забавные; пример см. на «The P-versus-NP page».

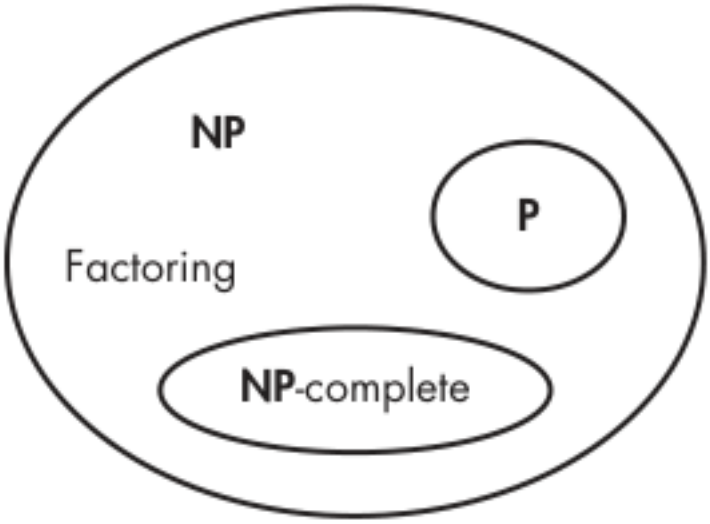

Рисунок 9-4. Классы NP и P и множество NP-полных задач

Если мы почти уверены, что в NP существуют трудные задачи, почему бы не использовать их для построения сильной криптографии с доказуемой безопасностью? Представьте доказательство того, что взлом некоторого шифра NP-труден, а значит, шифр невозможно взломать, пока P не равно NP. Но реальность разочаровывает: поисковые версии NP-полных задач трудно использовать в криптографии, потому что та самая структура, которая делает их трудными в худшем случае, может облегчать их в отдельных случаях, иногда возникающих в криптографии. Поэтому криптография часто опирается на задачи, которые, вероятно, не являются NP-трудными.

Задача факторизации

Задача факторизации состоит в нахождении простых чисел `p` и `q` по большому числу `N=p* q`. Широко применяемые алгоритмы RSA основаны на трудности факторизации: схемы шифрования и подписи RSA безопасны, потому что факторизация - трудная задача. Прежде чем в главе 10 мы увидим, как RSA использует факторизацию, я хотел бы убедить вас, что версия этой задачи в форме разрешимости - «Имеет ли `N` множитель меньше `k`, не равный 1?» - действительно трудна, но, вероятно, не является NP-полной.

Сначала немного базовой математики. Простое число не делится ни на какое другое число, кроме себя и 1. Например, 3, 7 и 11 - простые числа, а 4, то есть `2*2`, 6, то есть `2*3`, и 12, то есть `2*2*3`, - нет. Основная теорема арифметики утверждает, что любое целое число можно единственным образом записать как произведение простых чисел; такое представление называется факторизацией числа. Например, факторизация 123 456 равна `2^6*3*643`, факторизация 1 234 567 - `127*9721` и так далее. У каждого целого числа есть единственная факторизация, то есть единственный способ записи в виде произведения простых чисел. Алгоритмы проверки простоты полиномиального времени позволяют эффективно установить, является ли заданное число простым и содержит ли заданная факторизация только простые числа. Однако переход от числа к его простым множителям - совсем другое дело.

Факторизация больших чисел

Как же перейти от числа к его факторизации, то есть разложению в произведение простых чисел? Самый простой способ факторизовать число N - пробовать делить его на все меньшие числа, пока не найдется число x, делящее N. Затем попытаться разделить N на следующее число x+1 и так далее. В результате получится список множителей N. Какова временная сложность? Сначала вспомним, что сложность выражается как функция длины входных данных. Битовая длина числа N равна n=log_2N. По определению логарифма это означает N=2^n. Поскольку все числа меньше N/2 являются разумными кандидатами на роль множителей N, нужно испытать около N/2=2^n/2 значений. Поэтому сложность наивного алгоритма факторизации равна O(2^n), если в нотации O() отбросить коэффициент 1/2.

Многие числа легко факторизовать: сначала найти небольшие множители 2, 3, 5 и так далее, а затем итеративно факторизовать остальные непростые множители. Но нас интересуют встречающиеся в криптографии числа вида N=p* q, где p и q велики.

Поступим немного умнее: проверять все числа меньше N/2 не нужно, достаточно простых чисел меньше квадратного корня из N. Если N не является простым, у него должен быть хотя бы один множитель меньше sqrt(N). Если бы оба множителя N, p и q, были больше sqrt(N), их произведение было бы больше sqrt(N)timessqrt(N)=N, что невозможно. Например, если N=100, множители p и q не могут оба быть больше 10, поскольку тогда произведение превысило бы 100. Либо p, либо q обязано быть меньше или равно sqrt(N).

Какова сложность проверки только простых чисел меньше sqrt(N)? Теорема о распределении простых чисел утверждает, что простых чисел меньше N приблизительно N/log N. Следовательно, простых чисел меньше sqrt(N) приблизительно sqrt(N)/logsqrt(N). Выразив это значение через n=log_2N, получим примерно 2^(n/2+1)/n возможных простых множителей, а значит, сложность O(2^(n/2)/n), поскольку sqrt(N)=2^(n/2) и 1/logsqrt(N)=1/(n/2)=2/n. Это быстрее проверки всех простых чисел, но все равно мучительно медленно: порядка 2^120 операций для 256-битного числа. Такие вычислительные усилия совершенно непрактичны. Но можно намного лучше.

Самый быстрый алгоритм факторизации - общее решето числового поля (general number field sieve, GNFS). Я не буду описывать его здесь, поскольку для этого пришлось бы ввести несколько продвинутых математических понятий. Грубая оценка сложности GNFS равна

```
exp(1,91* n^(1/3)(log n)^(2/3)),
```

где n=log_2N - битовая длина N, а exp(...) - другая запись экспоненциальной функции e^x, причем экспоненциальная константа e приблизительно равна 2,718. Однако точно оценить фактическую сложность GNFS для заданного размера числа трудно. Поэтому приходится опираться на эвристические оценки сложности, показывающие рост безопасности с увеличением n. Например:

• Факторизация 1024-битного числа с двумя простыми множителями примерно по 500 битов требует порядка 2^70 базовых операций.
• Факторизация 2048-битного числа с двумя простыми множителями примерно по 1000 битов требует порядка 2^90 базовых операций - примерно в миллион раз больше, чем для 1024-битного числа.

Можно оценить, что для достижения 128-битной безопасности требуется не менее 4096 битов. Относитесь к этим значениям с осторожностью: исследователи не всегда сходятся в оценках. Следующие экспериментальные результаты показывают фактическую стоимость факторизации:

• В 2005 году примерно после 18 месяцев вычислений группа исследователей факторизовала 663-битное число, состоявшее из 200 десятичных цифр, благодаря мощности кластера из 80 процессоров. Общие усилия соответствовали 75 годам вычислений на одном процессоре.

• В 2009 году примерно после двух лет работы на нескольких сотнях процессоров другая группа исследователей факторизовала 768-битное число из 232 десятичных цифр. Общие усилия соответствовали приблизительно 2000 годам вычислений на одном процессоре.
• В 2020 году после нескольких месяцев вычислений на десятках тысяч процессоров и суперкомпьютере еще одна группа факторизовала 829-битное число из 250 десятичных цифр. Общие усилия соответствовали примерно 2700 годам вычислений на одном процессоре.

Как видите, факторизованные исследователями числа короче применяемых в реальных приложениях, где длина составляет не менее 1024, а часто более 2048 битов. На момент написания книги никто не сообщил о факторизации 1024-битного числа, но многие предполагают, что хорошо финансируемые организации вроде АНБ способны это сделать.

Итак, 1024-битный RSA следует считать небезопасным и использовать RSA со значением длиной не менее 2048 битов, а для более высокой безопасности предпочтительно 4096 битов.

Факторизация, вероятно, не является NP-трудной

Мы не знаем, как эффективно факторизовать большие числа, поэтому, вероятно, задача факторизации не принадлежит P. Однако версия факторизации в форме задачи разрешимости явно принадлежит NP: имея факторизацию, можно проверить решение, установив с помощью упомянутого алгоритма проверки простоты, что все множители просты, и убедившись, что их произведение дает ожидаемое число. Например, чтобы проверить, что `3*5` является факторизацией 15, нужно подтвердить простоту 3 и 5 и равенство произведения 3 и 5 числу 15.

Итак, у нас есть принадлежащая NP и выглядящая трудной задача. Но так ли она трудна, как самые трудные задачи NP? Иными словами, является ли версия факторизации в форме задачи разрешимости NP-полной и, следовательно, факторизация NP-трудной? Спойлер: вероятно, нет.

Математического доказательства того, что факторизация не NP-трудна, не существует, но есть несколько косвенных свидетельств. Во-первых, все известные NP-трудные задачи в NP могут иметь одно решение, несколько решений или не иметь ни одного. Факторизация, напротив, всегда имеет ровно одно решение. Кроме того, задача факторизации обладает математической структурой, позволяющей алгоритму GNFS значительно превосходить наивный алгоритм, тогда как у NP-трудных задач такой структуры нет. Факторизация была бы легкой для квантового компьютера - вычислительной модели, использующей явления квантовой механики для выполнения алгоритмов иного типа, способной эффективно факторизовать большие числа. Не потому, что такой компьютер быстрее выполнял бы обычный алгоритм, а потому, что он мог бы выполнить квантовый алгоритм, специально предназначенный для факторизации больших чисел. Однако такого квантового компьютера пока не существует и, возможно, никогда не будет. Считается, что для решения NP-трудных задач квантовый компьютер бесполезен, поскольку будет не быстрее классического; см. главу 14.

Теоретически факторизация может быть немного легче NP-трудных задач, но для криптографии она достаточно трудна и даже надежнее NP-трудных задач. Криптосистемы проще строить на задаче факторизации, чем на поисковых версиях NP-полных задач, поскольку для систем на основе таких задач трудно точно определить сложность взлома, то есть число битов безопасности. Как упоминалось выше, причина связана с тем, что NP относится к трудности худшего случая, тогда как криптографам нужна трудность в среднем случае.

Задача факторизации - одна из нескольких задач, используемых в криптографии как предположение о трудности, то есть предположение о вычислительной трудности некоторой задачи. Оно применяется при доказательстве того, что нарушение безопасности криптосистемы не легче решения данной задачи. Другая задача, используемая как предположение о трудности, -

задача дискретного логарифмирования (discrete logarithm problem, DLP), которая на самом деле является семейством задач.

Задача дискретного логарифмирования

В официальной истории криптографии задача дискретного логарифмирования появилась раньше задачи факторизации. RSA возникла в 1977 году, а другой криптографический прорыв - согласование ключей Диффи - Хеллмана, рассматриваемое в главе 11, - произошел годом раньше и основывал безопасность на трудности DLP. Подобно факторизации, DLP имеет дело с большими числами, но устроена менее очевидно и требует больше математики. Начнем с математического понятия группы в контексте дискретных логарифмов.

Группы

В математике группа - множество элементов, обычно чисел, связанных друг с другом некоторыми четко определенными правилами. Пример группы - множество ненулевых целых чисел по модулю простого числа p, то есть чисел от 1 до p-1, с модульным умножением в качестве групповой операции. Такая группа обозначается (Z_p^*,*) или просто Z_p^*, когда ясно, что групповой операцией является умножение.

При p=5 получается группа Z_5^*={1,2,3,4}. В Z_5^* операции выполняются по модулю 5, поэтому вместо 3*4=12 получается 3*4=2, поскольку 12 mod 5=2. Тем не менее можно использовать тот же знак *, что для обычного умножения целых чисел. Степенная запись также применяется для обозначения умножения элемента группы на самого себя по модулю p - распространенной операции в криптографии. Например, в контексте Z_5^* значение 2^3=2*2*2=3, а не 8, поскольку 8 mod 5=3.

Чтобы математическое множество и его операция образовывали группу, они должны обладать следующими свойствами, называемыми аксиомами группы:

Замкнутость. Для любых двух элементов группы x и y значение x* y также принадлежит группе. В Z_5^* имеем 2*3=1, поскольку 6=1 mod 5, 2*4=3 и так далее.

Ассоциативность. Для любых элементов группы x, y, z выполняется (x* y)* z=x*(y* z). В Z_5^* имеем (2*3)*4=1*4=2*(3*4)=2*2=4.

Существование нейтрального элемента. Группа содержит элемент e, такой что e* x=x* e=x. Такой элемент называется нейтральным. В любой Z_p^* нейтральный элемент равен 1.

Существование обратного элемента. Для любого x в группе существует y, такой что x* y=y* x=e. В Z_5^* обратным к 2 является 3, а обратным к 3 - 2; число 4 обратно самому себе, поскольку 4*4=16=1 mod 5.

Кроме того, группа называется коммутативной, или абелевой, если x* y=y* x для любых ее элементов x и y. Это верно для любой мультипликативной группы целых чисел Z_p^*. В частности, Z_5^* коммутативна: 3*4=4*3, 2*3=3*2 и так далее.

Группа называется циклической, если существует хотя бы один элемент g, степени которого g^1,g^2,g^3 и так далее по модулю p охватывают все различные элементы группы. Тогда элемент g называется генератором группы. Z_5^* циклическая и имеет два генератора, 2 и 3, поскольку 2^1=2, 2^2=4, 2^3=3, 2^4=1, а 3^1=3, 3^2=4, 3^3=2, 3^4=1.

Здесь в качестве групповой операции используется умножение, но группы можно получить и с другими операциями. Например, самая очевидная группа - множество всех положительных и отрицательных целых чисел со сложением в качестве групповой операции. Проверим аксиомы группы со сложением в приведенном порядке: x+y является целым, если x и y целые, - замкнутость; (x+y)+z=x+(y+z) для любых x, y и z - ассоциативность; ноль является нейтральным элементом; обратный к любому x элемент равен -x, поскольку x+(-x)=0 для любого целого x. Существенное различие в том, что эта группа целых чисел бесконечна, тогда как в криптографии по причинам реализации используются только конечные группы, то есть группы с конечным числом элементов. Обычно это группы Z_p^*, где длина p составляет тысячи битов, то есть группы содержат порядка 2^m чисел, если длина p равна m битам.

Что здесь трудно

Задача дискретного логарифмирования в первоначальном криптографическом применении состоит в нахождении y, для которого g^y=x, при заданном генераторе g некоторой группы Z_p^*, где p - простое число, и заданном элементе группы x. DLP часто записывается в аддитивной, а не мультипликативной форме, как для групп эллиптических кривых. Тогда требуется найти множитель k, такой что k* P=Q, при известных точках P и Q. Эта задача называется DLP на эллиптической кривой (elliptic curve DLP, ECDLP).

DLP является дискретной, поскольку мы имеем дело со счетными целыми, а не несчетными действительными числами, и логарифмической, поскольку ищем логарифм x по основанию g. Например, логарифм 256 по основанию 2 равен 8, потому что 2^8=256.

Факторизация примерно столь же трудна и, следовательно, столь же безопасна, как дискретное логарифмирование. Алгоритмы решения DLP похожи на алгоритмы факторизации целых чисел, а n-битные труднофакторизуемые числа дают примерно тот же уровень безопасности, что дискретные логарифмы в n-битной группе. По той же причине, что факторизация, DLP не является NP-трудной. Существуют группы, где DLP решить легче, но в криптографии они не используются, по крайней мере там, где нужна трудность DLP.

Что может пойти не так

Более чем через 40 лет мы все еще не знаем, как эффективно факторизовать большие числа или решать дискретные логарифмы. Без математического доказательства всегда можно предположить, что однажды они будут взломаны. Но у нас нет и доказательства P!= NP, поэтому можно предположить P=NP; однако специалисты считают такой сюрприз маловероятным. Большинство развернутой сегодня криптографии с открытым ключом опирается либо на факторизацию, RSA, либо на DLP: Диффи - Хеллман, ElGamal, криптография на эллиптических кривых. Возможно, математика нас не подведет, но могут вмешаться практические обстоятельства и человеческие ошибки.

Когда факторизация проста

Факторизация больших чисел трудна не всегда. Рассмотрим, например, следующее 1024-битное число N:

```
179769313486231590772930519078902473361797697894230657273430081157739343819933
842986982557174198257278917258638193709265819186026626180659730665062710995556
578639447715608415186895652841691982921107202317165369124890481512388558039053
427125099290315449262324709315263256083132540461407052872832790915388014592
```

Для 1024-битных чисел в схемах шифрования или подписи RSA, где `N=pq`, ожидается, что лучшим алгоритмам факторизации потребуется около `2^70` операций, как обсуждалось выше. Но этот пример можно факторизовать за секунды с помощью SageMath - математической программы на основе Python. Функция `factor()` SageMath на моем MacBook 2023 года нашла следующую факторизацию менее чем за секунду:

```
2^800*641*6700417*167773885276849215533569
()*37414057161322375957408148834323969.
```

Да, я схитрил. Это число не имеет вида `N=pq`, поскольку у него не два больших простых множителя, а пять, включая очень малые, поэтому его легко факторизовать. Сначала часть `2^800*641*6 700 417` обнаруживается перебором малых простых чисел из заранее вычисленного списка, после чего остается 192-битное число, факторизовать которое намного легче, чем 1024-битное число с двумя большими множителями.

Факторизация может быть легкой не только при наличии у `N` малых простых множителей, но и когда `N` или его множители `p` и `q` имеют особую форму, например когда `N=pq`, причем и `p`, и `q` близки к некоторому `2^b`; когда `N=pq` и известны некоторые биты `p` или `q`; либо когда `N` имеет вид `N=p^rq^s`, а `r` больше `log p`. Однако подробное объяснение причин этих слабостей слишком техническое для этой книги.

В результате алгоритмы шифрования и подписи RSA, см. главу 10, должны работать со значением `N=pq`, где `p` и `q` тщательно выбраны, чтобы избежать легкой факторизации `N`, способной привести к катастрофе безопасности.

Малые трудные задачи не трудны

Вычислительно трудные задачи и даже алгоритмы экспоненциального времени становятся практичными, если достаточно малы. Симметричный шифр может быть безопасен в том смысле, что атаки быстрее полного перебора за `2^n` не существует, но при длине ключа `n=32` шифр будет взломан за минуты. Это кажется очевидным, и можно подумать, что никто не станет настолько наивно использовать короткие ключи, однако в действительности такое может происходить по множеству причин. Вот две реальные истории.

Предположим, вы разработчик, ничего не знающий о криптографии, но имеющий API для шифрования RSA, и вам велели обеспечить 128-битную безопасность. Какой размер ключа RSA вы выберете? Я встречал реальные случаи 128-битного RSA, то есть RSA на основе 128-битного числа `N=pq`. Но хотя факторизация практически неосуществима для `N` длиной в тысячи битов, 128-битное число факторизовать легко. Следующие команды SageMath в листинге 9-2 выполняются мгновенно.

```
sage: p = random_prime(2**64)
sage: print(p)
6822485253121677229
sage: q = random_prime(2**64)
sage: print(q)
17596998848870549923
Sage: N = p*q
sage: factor(N)
6822485253121677229 * 17596998848870549923
```

Листинг 9-2. Создание модуля RSA путем выбора двух случайных простых чисел и его мгновенная факторизация

Листинг 9-2 показывает, что на обычном ноутбуке легко факторизовать случайное 128-битное число, являющееся произведением двух 64-битных простых. Но если вместо этого выбрать 1024-битные простые числа с помощью `p = random_prime(2**1024)`, команда `factor(p*q)` никогда не завершится, по крайней мере при моей жизни.

Справедливости ради, доступные инструменты не помогают предотвращать наивное применение небезопасно коротких параметров. Например, инструментарий OpenSSL прежде позволял без всякого предупреждения создавать ключи RSA длиной всего 31 бит; такие короткие ключи совершенно небезопасны, как показано в листинге 9-3. С тех пор OpenSSL был исправлен, и версия 1.1.1t от февраля 2023 года возвращает ошибку `key size too small`, если запросить ключ короче 512 битов.

```
$ openssl genrsa 31
Generating RSA private key, 31 bit long modulus
.++++++++++++++++++++++++++++
.++++++++++++++++++++++++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBHHqFuUCAwEAAQIEP6zEJQIDANATAgMAjCcCAwCSBwICTGsCAhpp
-----END RSA PRIVATE KEY-----
```

Листинг 9-3. Создание небезопасного закрытого ключа RSA старой версией инструментария OpenSSL

При проверке криптографии нужно учитывать не только тип используемых алгоритмов, но и их параметры и длину секретных значений. Однако, как покажет следующая история, то, что достаточно безопасно сегодня, завтра может стать небезопасным.

В 2015 году исследователи обнаружили, что многие HTTPS- и почтовые серверы поддерживали старую небезопасную версию протокола согласования ключей Диффи - Хеллмана. В частности, лежащая в основе реализация TLS поддерживала Диффи - Хеллман в группе `Z_p^*`, определенной простым числом `p` длиной всего 512 битов, где задача дискретного логарифмирования уже не была практически неразрешимой.

Серверы не только поддерживали слабый алгоритм: атакующие могли принудить обычного клиента использовать его, внедрив вредоносный трафик в сеанс клиента. Что еще лучше для атакующих, основную часть атаки можно было выполнить один раз и повторно использовать против множества клиентов. После приблизительно недели вычислений для атаки на конкретную группу `Z_p^*` взлом отдельного сеанса любого пользователя занимал всего 70 секунд.

Безопасный протокол ничего не стоит, если его подрывает ослабленный алгоритм, а надежный алгоритм бесполезен, если саботирован слабыми параметрами. В криптографии всегда нужно читать мелкий шрифт.

Подробности этой истории см. в исследовательской статье «Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice».

Дополнительная литература

Я призываю глубже изучить основополагающие аспекты вычислений в контексте вычислимости - какие функции можно вычислить? - и сложности - с какой стоимостью? - а также их связь с криптографией. Я говорил в основном о классах P и NP, но существует множество других классов и тем, интересных криптографам. Настоятельно рекомендую книгу Скотта Аронсона «Quantum Computing Since Democritus» (Cambridge University Press, 2013). Она в значительной степени посвящена квантовым вычислениям, но первые главы блестяще знакомят с теорией сложности и криптографией.

В исследовательской литературе по криптографии встречаются и другие трудные вычислительные задачи. Я упомяну их в последующих главах, а здесь приведу несколько примеров, показывающих разнообразие задач, используемых криптографами:

• Задача Диффи - Хеллмана - по заданным g^x и g^y найти g^xy - является вариантом задачи дискретного логарифмирования и широко применяется в протоколах согласования ключей.
• Задачи на решетках, такие как задача поиска кратчайшего вектора (shortest vector problem, SVP), задача коротких целочисленных решений (short integer solutions, SIS) и обучение с ошибками (learning with errors, LWE), в зависимости от параметров могут быть NP-трудными.
• Задачи теории кодирования основаны на трудности декодирования кодов с исправлением ошибок при недостаточной информации и исследуются с конца 1970-х годов. Они также могут быть NP-трудными.
• Многомерные задачи связаны с решением нелинейных систем уравнений и потенциально являются NP-трудными, однако создать на их основе надежные криптосистемы не удалось: трудные версии слишком велики и медленны, а практические оказались небезопасны.

В главе 10 мы продолжим изучать трудные задачи, особенно факторизацию и ее основной вариант - задачу RSA.

Глава 10. RSA

Криптосистема Ривеста - Шамира - Адлемана (RSA) произвела революцию в криптографии, появившись в 1977 году как первая схема шифрования с открытым ключом. Классические схемы шифрования с симметричным ключом используют один секретный ключ для шифрования и расшифрования сообщений, а шифрование с открытым ключом, или асимметричное шифрование, использует два ключа: открытый, которым может воспользоваться любой желающий зашифровать для вас сообщение, и закрытый, необходимый для расшифрования сообщений, зашифрованных открытым ключом. Эта магия сделала RSA настоящим прорывом, а 40 лет спустя он по-прежнему остается образцом шифрования с открытым ключом и рабочей лошадкой безопасности интернета. За год до RSA Уитфилд Диффи и Мартин Хеллман представили понятие криптографии с открытым ключом, но их схема могла выполнять в такой среде только распределение ключей.

RSA работает, создавая перестановку с потайным входом - функцию, преобразующую число x в число y из того же диапазона так, что вычислить y из x с помощью открытого ключа легко, а вычислить x из y практически невозможно без знания закрытого ключа, то есть потайного входа. Считайте x открытым текстом, а y - шифртекстом.

Помимо шифрования, RSA позволяет строить цифровые подписи, при которых подписать сообщение способен только владелец закрытого ключа, а открытый ключ дает любому возможность проверить действительность подписи.

В этой главе вы узнаете, как работает перестановка RSA с потайным входом и почему одной этой перестановки недостаточно для безопасного шифрования и подписи. Я также рассмотрю безопасность RSA относительно задачи факторизации из главы 9, способы безопасной реализации RSA и атаки на него.

Начнем с объяснения базовых математических понятий, лежащих в основе RSA.

Математика RSA

При обработке сообщения, будь то шифрование или подписание, RSA сначала создает из него большое число, а затем обрабатывает это число, выполняя умножения больших чисел. Поэтому для понимания RSA нужно знать, с какими большими числами он работает и как выполняется их умножение.

Для шифрования или подписи RSA преобразует положительное целое число от 1 до n-1, где n - большое число, называемое модулем. Произведение таких чисел дает другое число, удовлетворяющее тем же условиям. Эти числа образуют группу Z_n^*, называемую мультипликативной группой целых чисел по модулю n. Математическое определение группы см. в разделе «Группы».

Рассмотрим, например, группу целых чисел по модулю 4, Z_4^*. Напомним из главы 9, что группа должна содержать нейтральный элемент, то есть 1, а у каждого числа x в группе должен существовать обратный элемент y, такой что x* y=1. Как определить множество, образующее Z_4^*? Из определений следует, что 0 не принадлежит Z_4^*, поскольку умножение любого числа на 0 никогда не даст 1, а значит, у 0 нет обратного элемента. Число 1 принадлежит Z_4^*, потому что 1*1=1, то есть 1 обратно самому себе. Однако 2 группе не принадлежит, поскольку умножением 2 на другой элемент Z_4^* получить 1 невозможно. Обратите внимание, что 2 не взаимно просто с 4, поскольку у 4 и 2 есть общий множитель 2. Число 3 принадлежит Z_4^*, поскольку обратно самому себе внутри Z_4^*. Таким образом, Z_4^*={1,3}.

Теперь рассмотрим Z_5^* - мультипликативную группу целых чисел по модулю 5. Как и в главе 9, Z_5^*={1,2,3,4}. Поскольку 5 простое, числа 1, 2, 3 и 4 взаимно просты с 5 и все входят в Z_5^*. Проверим: 2*3 mod 5=1, поэтому 2 является обратным к 3, а 3 - обратным к 2; 4 обратно самому себе, поскольку 4*4 mod 5=1; наконец, 1 снова обратно самому себе в группе.

Чтобы найти число элементов группы Z_n^*, когда n не простое, можно использовать функцию Эйлера, записываемую phi(n), где phi - греческая буква фи. Эта функция дает число элементов, взаимно простых с n, то есть число элементов Z_n^*. Как правило, если n является произведением простых чисел n=p_1* p_2*...* p_m, число элементов группы Z_n^* равно

```
phi(n)=(p_1-1)*(p_2-1)*...*(p_m-1).
```

RSA работает только с числами n, являющимися произведением двух больших простых, обычно записываемым n=pq. Тогда соответствующая группа Z_n^* содержит phi(n)=(p-1)(q-1) элементов. Раскрыв скобки, получаем эквивалентное определение phi(n)=n-p-q+1, или phi(n)=(n+1)-(p+q), которое нагляднее выражает значение phi(n) относительно n.

Перестановка RSA с потайным входом

Перестановка RSA с потайным входом - основной алгоритм шифрования и подписи на основе RSA. Имея модуль n и число e, называемое открытым показателем, перестановка RSA преобразует число x из множества Z_n^* в число y=x^e mod n. Иными словами, вычисляется значение, равное x, умноженному на себя e-1 раз по модулю n, и возвращается результат. Когда перестановка RSA используется для шифрования, модуль n и показатель e образуют открытый ключ RSA.

Чтобы восстановить x из y, можно использовать другое число d и вычислить

```
y^d mod n=(x^e)^d mod n=x^ed mod n=x.
```

Поскольку d является потайным входом, позволяющим расшифровывать, он входит в закрытый ключ пары RSA и всегда должен оставаться секретным. Число d также называется секретным показателем.

Разумеется, d - не произвольное число, а такое, что произведение e и d эквивалентно 1 и потому x^ed mod n=x для любого x. Точнее, должно выполняться ed mod phi(n)=1, чтобы получить x^ed=x^1=x и правильно расшифровать сообщение. Обратите внимание, что здесь вычисление выполняется по модулю phi(n), а не n, поскольку показатели ведут себя как индексы элементов Z_n^*, а не как сами элементы. Поскольку Z_n^* содержит phi(n) элементов, индекс должен быть меньше phi(n).

Число phi(n) критически важно для безопасности RSA. Нахождение phi(n) для модуля RSA n фактически равносильно взлому RSA, поскольку из phi(n) и e легко получить секретный показатель d, вычислив обратный к e элемент. Поэтому p и q также должны оставаться секретными: знание p или q дает phi(n) путем вычисления (p-1)(q-1)=phi(n).

> **Примечание.** phi(n) называется порядком группы Z_n^*. Порядок - важная характеристика группы, необходимая и другим системам с открытым ключом, таким как Диффи - Хеллман и криптография на эллиптических кривых.

Генерация ключей RSA и безопасность

Генерация ключей - процесс создания пары ключей RSA: открытого ключа, состоящего из модуля `n` и открытого показателя `e`, и соответствующего закрытого ключа с секретным показателем `d`. Поскольку числа `p` и `q`, такие что `n=pq`, а также порядок `phi(n)` должны оставаться секретными, их часто включают в закрытый ключ.

Чтобы создать пару ключей RSA, сначала выбираются два случайных простых числа `p` и `q` и по ним вычисляется `phi(n)`. Затем `d` вычисляется как обратное к `e`. В листинге 10-1 показано, как это делается в SageMath - среде с открытым исходным кодом, похожей на Python и включающей множество математических пакетов, см. https://www.sagemath.org.

```
sage: p = random_prime(2^32); p
1103222539
sage: q = random_prime(2^32); q
17870599
sage: n = p*q; n
19715247602230861
sage: phi = (p-1)*(q-1); phi
19715246481137724
sage: e = random_prime(phi); e
13771927877214701
sage: d = xgcd(e, phi); d
15417970063428857
sage: mod(d*e, phi)
1
```

Листинг 10-1. Генерация параметров RSA с помощью SageMath

Функция `random_prime()` выбирает случайные простые `p` и `q`, меньшие заданного аргумента. Затем `p` и `q` перемножаются для получения модуля `n` и `phi(n)`, представленного переменной `phi`. После этого генерируется случайный открытый показатель `e`: выбирается случайное простое меньше `phi`, чтобы у `e` существовал обратный элемент по модулю `phi`.

Связанный закрытый показатель `d` формируется функцией Sage `xgcd()`. Эта функция при помощи расширенного алгоритма Евклида вычисляет для двух чисел `a` и `b` числа `s` и `t`, такие что `as+bt=GCD(a,b)`. Наконец, проверяется равенство `ed mod phi(n)=1`, подтверждающее, что `d` правильно обращает перестановку RSA.

> **Примечание.** В листинге 10-1 использован 64-битный модуль `n`, чтобы вывод не занял несколько страниц, но на практике для безопасности модуль RSA должен иметь длину не менее 2048 битов.

Теперь можно применить перестановку с потайным входом, как показано в листинге 10-2.

```
sage: x = 1234567
sage: y = power_mod(x, e, n); y
19048323055755904
sage: power_mod(y, d, n)
1234567
```

Листинг 10-2. Прямое и обратное вычисление перестановки RSA с потайным входом

Целое число 1 234 567 присваивается `x`, а затем функция `power_mod(x, e, n)`, возведение в степень по модулю `n`, то есть `x^e mod n`, вычисляет `y`. После вычисления `y=x^e mod n` значение `y^d mod n` вычисляется с потайным входом `d` и возвращает исходное `x`.

Насколько трудно найти `x` без потайного входа `d`? Атакующий, способный факторизовать большие числа, может взломать RSA, восстановив `p` и `q`, затем `phi(n)` и вычислив `d` из `e`. Другая угроза RSA - способность атакующего вычислять `x` из `x^e mod n`, то есть корни степени `e` по модулю `n`, необязательно факторизуя `n`. Хотя обе угрозы кажутся тесно связанными, их эквивалентность достоверно неизвестна.

Если предположить, что факторизация и нахождение корней степени e примерно одинаково трудны, уровень безопасности RSA зависит от трех факторов: размера n, выбора p и q и способа применения перестановки с потайным входом. Если n слишком мало, его можно факторизовать за реальное время и раскрыть закрытый ключ. Для безопасности длина n должна составлять не менее 2048 битов, что дает около 90 битов безопасности и требует примерно 2^90 операций, но предпочтительно 4096 битов, что дает около 128 битов безопасности. Значения p и q должны быть несвязанными случайными простыми числами сходного размера. Если они слишком малы или слишком близки друг к другу, определить их из n легче. Наконец, перестановку RSA с потайным входом нельзя напрямую использовать для шифрования или подписи, как будет показано далее.

Шифрование с RSA

Обычно RSA применяется вместе со схемой симметричного шифрования: он шифрует симметричный ключ, которым сообщение шифруется симметричным шифром, например AES-GCM. Но шифрование сообщения или симметричного ключа с RSA сложнее простого преобразования цели в число x и вычисления x^e mod n.

В следующих подразделах я объясню, почему наивное применение перестановки RSA небезопасно и как работает стойкое шифрование на основе RSA.

Податливость учебного шифрования RSA

Выражение «учебное шифрование RSA» обозначает простейшую схему RSA, где возводимое в степень число содержит только сообщение, которое требуется зашифровать. Например, чтобы зашифровать строку RSA, сначала ее преобразуют в число, например сцепляя ASCII-коды каждой из трех букв в виде байтов: R (байт 52), S (байт 53) и A (байт 41). Полученная строка байтов 525341 при преобразовании в десятичное число равна 5 395 265, которое затем можно зашифровать, вычислив 5 395 265^e mod n. Не зная секретного ключа, расшифровать сообщение невозможно - по крайней мере, в теории.

Однако учебное шифрование RSA детерминировано: если дважды зашифровать один и тот же открытый текст, дважды получится один и тот же шифртекст. Есть и еще более серьезная проблема: имея два шифртекста учебного RSA y_1=x_1^e mod n и y_2=x_2^e mod n, можно получить шифртекст для x_1* x_2, перемножив эти шифртексты:

```
y_1* y_2 mod n=x_1^e* x_2^e mod n=(x_1* x_2)^e mod n.
```

Результат равен (x_1* x_2)^e mod n, то есть шифртексту сообщения x_1* x_2 mod n. Атакующий может создать новый действительный шифртекст из двух шифртекстов RSA, нарушив безопасность шифрования благодаря возможности получить сведения об исходном сообщении. Из-за этой слабости учебное шифрование RSA является податливым. (Если известны x_1 и x_2, то также можно вычислить (x_1* x_2)^e mod n, однако, зная только y_1 и y_2, вы не должны иметь возможности перемножить шифртексты и получить шифртекст произведения открытых текстов.)

Еще одна проблема наивного учебного шифрования RSA состоит в существовании «особых» сообщений. Какими бы ни были n и e, 1^e=1. Поэтому сообщение 1 при шифровании не изменяется. У учебного RSA есть много других проблем, но вы научитесь избегать их, используя стойкое шифрование RSA.

Стойкое шифрование RSA с OAEP

Чтобы сделать шифртексты RSA неподатливыми, число, возводимое в степень при шифровании, должно объединять данные сообщения с дополнительными данными, называемыми дополнением, как показано на рисунке 10-1.

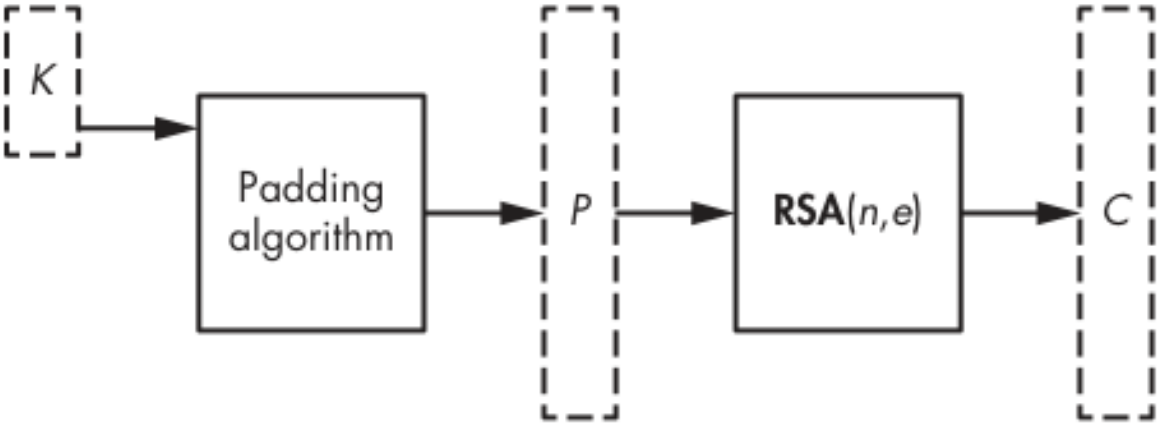

Рисунок 10-1. Шифрование симметричного ключа `K` с помощью RSA, где открытым ключом служит пара `(n,e)`

Стандартный способ такого шифрования с RSA использует оптимальное асимметричное дополнение шифрования (Optimal Asymmetric Encryption Padding, OAEP); эту комбинацию обычно называют RSA-OAEP. В этой схеме перед применением функции RSA создается битовая строка размером с модуль: сообщение дополняется дополнительными данными и случайными значениями.

> **Примечание.** В официальных документах, таких как стандарт PKCS#1 и специальная публикация NIST 800-56B, OAEP называется RSAES-OAEP. OAEP усовершенствовало более ранний метод
>
> PKCS#1 v1.5 - один из первых в серии стандартов криптографии с открытым ключом (Public-Key Cryptography Standards, PKCS), созданных RSA. Он заметно менее безопасен, чем OAEP, однако после появления OAEP продолжал использоваться во многих системах.

Безопасность

OAEP использует ГПСЧ, чтобы обеспечить неразличимость и неподатливость шифртекстов, сделав шифрование вероятностным. Доказано, что схема безопасна при условии безопасности функции RSA и ГПСЧ, а также, в меньшей степени, при условии, что хеш-функции не слишком слабы. При шифровании с RSA следует применять OAEP, а не его предшественника, стандарт PKCS#1 v1.5.

Шифрование

Для шифрования с RSA в режиме OAEP нужны сообщение (например, симметричный ключ `K`), ГПСЧ и две хеш-функции. Чтобы создать шифртекст, возьмем заданный модуль `n` длиной `m` байтов (то есть `8m` битов, а значит, `n` меньше `2^8m`). Для шифрования `K` сформируем кодированное сообщение

```
M=H || 00... 00 || 01 || K,
```

где H - определенная схемой OAEP константа длиной h байтов, за которой следует столько байтов 00, сколько необходимо, а затем байт 01. Обработка этого кодированного сообщения M показана на рисунке 10-2.

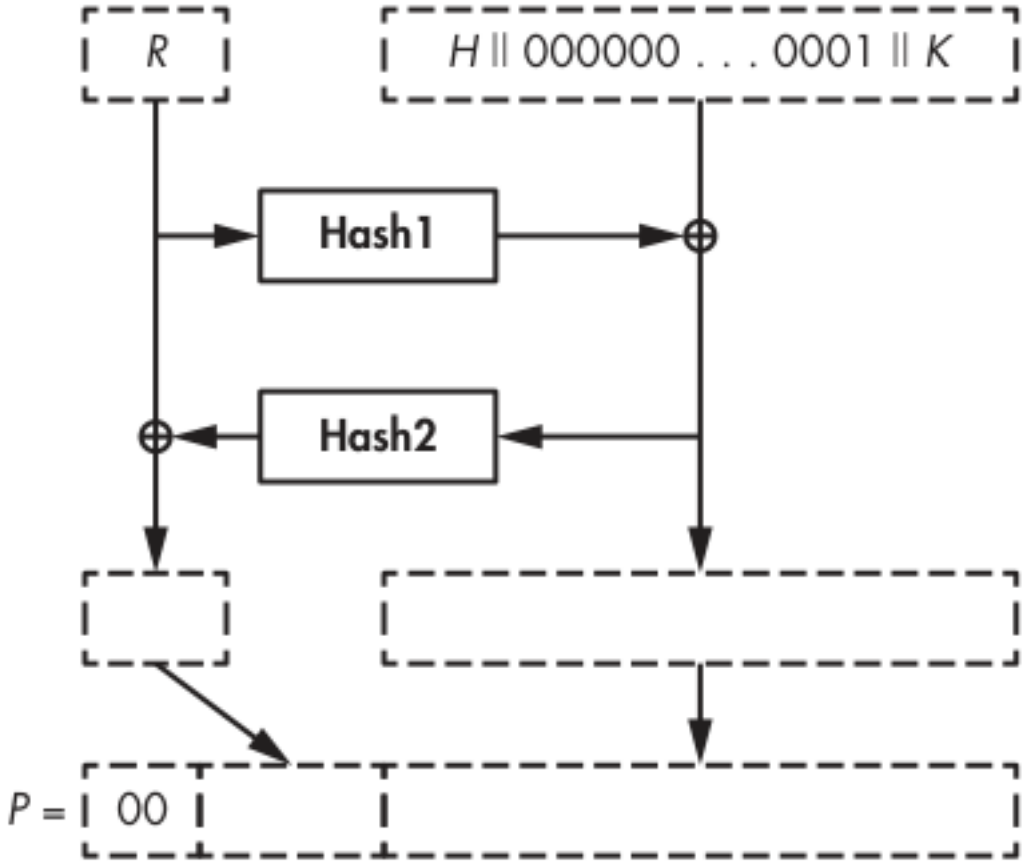

Рисунок 10-2. Шифрование симметричного ключа K с помощью RSA-OAEP, где H - фиксированный параметр, а R - случайные биты

Вы генерируете случайную строку R длиной h байтов и задаете M=MoplusHash1(R), где Hash1(R) имеет ту же длину, что и M. Затем задаете R=RoplusHash2(M), где Hash2(M) имеет ту же длину, что и R. Теперь из новых значений M и R формируется строка P=00 || M || R длиной m байтов, равной длине модуля n, которую можно преобразовать в целое число меньше n. Результатом такого преобразования является число x, после чего функция RSA x^e mod n дает шифртекст.

Чтобы расшифровать шифртекст y, сначала вычисляют x=y^d mod n и восстанавливают из результата конечные значения M и R. Затем извлекают исходное значение M, вычисляя MoplusHash1(RoplusHash2(M)). Наконец, проверяют, что M имеет вид H || 00... 00 || 01 || K: H длиной h байтов, за которым следуют байты 00, а затем байт 01.

На практике параметрам m и h (соответственно длине модуля и длине результата Hash2) обычно присваивают значения m=256 байтов (для 2048-битного RSA) и h=32 (при использовании SHA-256 в качестве Hash2). Тогда для M остается m-h-1=223 байта, из которых для K доступны не более m-2h-2=190 байтов («-2» обусловлено байтом-разделителем 01 в M). Хеш-значение Hash1 в этом случае состоит из m-h-1=223 байтов, что длиннее хеш-значения любой распространенной хеш-функции. Для построения хеша с такой необычной длиной результата спецификация стандарта RSA определяет метод функции генерации маски, позволяющий на основе любой хеш-функции создавать хеш-функции, возвращающие хеш-значения произвольной длины. Другой вариант - использовать функцию с расширяемым выводом (XOF), например SHAKE или BLAKE3,

хотя это и отличается от стандартных спецификаций.

Подпись с помощью RSA

Цифровые подписи могут доказывать, что некоторое сообщение подписал владелец закрытого ключа, связанного с конкретной цифровой подписью, обычно в знак одобрения его содержимого. Поскольку закрытый показатель d известен только владельцу закрытого ключа, никто другой не может вычислить подпись y=x^d mod n некоторого значения x, однако любой может проверить равенство y^e mod n=x, имея открытый показатель e. В принципе, такую проверенную подпись можно использовать как свидетельство того, что владелец закрытого ключа подписал конкретное сообщение; это свойство называется неотказуемостью.

Соблазнительно считать подписи RSA обратной стороной шифрования, но это неверно. Подпись с помощью RSA - не то же самое, что шифрование закрытым ключом. Шифрование обеспечивает конфиденциальность, тогда как цифровые подписи помогают предотвращать подделки. Самый показательный пример этого различия: схема подписи вполне может раскрывать сведения о подписанном сообщении, поскольку сообщение не является секретным. Например, схема, раскрывающая части сообщений, может быть безопасной схемой подписи, но не безопасной схемой шифрования.

Из-за неизбежных накладных расходов шифрование с открытым ключом способно обрабатывать лишь короткие сообщения - обычно секретные ключи, а не собственно сообщения. Однако схема подписи может обрабатывать сообщения произвольного размера, используя вместо них хеш-значения Hash(M), и при этом оставаться безопасной, даже будучи детерминированной. Как и RSA-OAEP, схемы подписи на основе RSA могут применять схему дополнения, но также могут использовать максимальное пространство сообщений, допускаемое модулем RSA.

Учебные подписи RSA

Учебной подписью RSA называется метод, который подписывает сообщение x, непосредственно вычисляя y=x^d mod n, где x может быть любым числом от 1 до n-1. Как и учебное шифрование, учебная подпись RSA проста в описании и реализации, но небезопасна перед лицом нескольких атак. Одна из них допускает тривиальную подделку: заметив, что 1^d mod n=1 и (n-1)^d mod n=n-1 независимо от значения закрытого ключа d, атакующий может подделать подписи чисел 1 и n-1, не зная d.

Еще опаснее атака с ослеплением. Допустим, вы хотите получить подпись третьего лица под некоторым сообщением M, которое оно никогда не стало бы подписывать сознательно. Для проведения атаки сначала найдите значение R, при котором R^eM mod n является сообщением, которое жертва согласится подписать. Затем убедите ее подписать это сообщение и показать вам подпись, равную S=(R^eM)^d mod n, то есть сообщению, возведенному в степень d. Имея эту подпись, можно с помощью несложных вычислений получить подпись M, а именно M^d.

Поскольку S можно записать как (R^eM)^d=R^edM^d и, по определению, R^ed=R, имеем S=(R^eM)^d=RM^d. Чтобы получить подпись M^d, разделим S на R:

```
S/R=RM^d/R=M^d.
```

Часто это практичная и мощная атака.

Стандарт подписи PSS

Вероятностная схема подписи RSA (RSA Probabilistic Signature Scheme, PSS) для подписей RSA является тем же, чем OAEP для шифрования RSA. Она была разработана, чтобы повысить безопасность подписи сообщений за счет добавления данных дополнения.

На рисунке 10-3 показано, что PSS объединяет сообщение, более узкое, чем модуль, с некоторыми случайными и фиксированными битами, прежде чем применить RSA к результатам этого процесса дополнения.

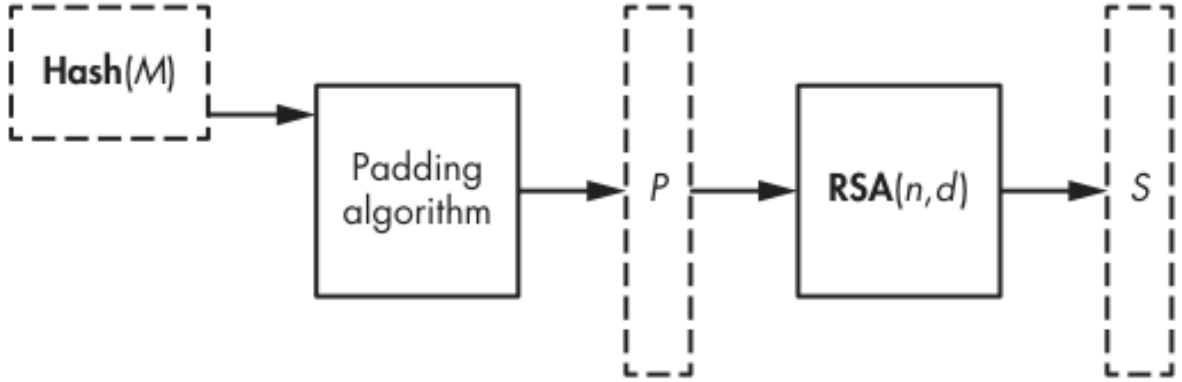

Рисунок 10-3. Подпись сообщения `M` с помощью RSA и стандарта PSS, где `(n,d)` - закрытый ключ

Как и все схемы подписи с открытым ключом, PSS работает с хешем сообщения, а не с самим сообщением. Подпись `Hash(M)` безопасна только в том случае, если хеш-функция устойчива к коллизиям. Поэтому можно подписывать сообщения любой длины: после хеширования сообщения получается хеш-значение одной и той же фиксированной длины независимо от исходной длины сообщения.

Почему бы просто не подписывать, применяя шифрование OAEP к `Hash(M)`? К сожалению, так нельзя. Хотя OAEP похожа на PSS, ее безопасность доказана только для шифрования, но не для подписей.

Как и OAEP, PSS также требует ГПСЧ и две хеш-функции. Первая, Hash1, - обычная хеш-функция со стандартной длиной результата, например SHA-256. Вторая, Hash2, - хеш-функция с широким выводом, подобная Hash2 в OAEP. Как и OAEP, PSS может использовать конструкцию функции генерации маски (mask-generating function, MGF) для построения такого хеша.

Процедура создания подписи PSS выполняется следующим образом (где `h` - длина результата Hash1):

1. С помощью ГПСЧ выбирается случайная строка `R` длиной `r` байтов.
2. Формируется кодированное сообщение `M'=0000000000000000 || Hash1(M) || R` длиной `h+r+8` байтов (с восемью нулевыми байтами в начале).
3. Вычисляется строка `H=Hash1(M')` длиной `h` байтов.
4. Задается `L=00... 00 || 01 || R`, то есть последовательность байтов 00, за которой следуют байт 01 и затем `R`; число байтов 00 выбирается так, чтобы длина `L` составляла `m-h-1` байтов (байтовая ширина `m` модуля минус длина хеша `h` минус 1).
5. Задается `L=LoplusHash2(H)`, то есть прежнее значение `L` заменяется новым.
6. Строка `P=L || H || BC` длиной `m` байтов преобразуется в число `x`, меньшее `n`. Здесь байт BC - фиксированное значение, добавляемое после `H`.
7. Для только что полученного значения `x` вычисляется функция RSA `x^d mod n`, дающая подпись.
8. Чтобы проверить подпись заданного сообщения `M`, вычисляют `Hash1(M)` и используют открытый показатель `e`, чтобы обратить функцию RSA и извлечь из подписи `L`, `H`, а затем `M'`, на каждом шаге

проверяя правильность дополнения.

На практике случайная строка `R` (в стандарте RSA-PSS она называется солью `t`) обычно имеет ту же длину, что и хеш-значение. Например, если используется `n=2048` битов и SHA-256, значение `L` занимает `m-h-1=256-32-1=223` байта, а случайная строка `R` обычно состоит из 32 байтов.

Как и OAEP, PSS имеет доказанную безопасность, стандартизирована и доступна во многих криптографических программах и библиотеках, включая OpenSSL и криптографический модуль языка Go. И подобно OAEP, она выглядит излишне сложной, из-за чего подвержена ошибкам реализации и неправильной обработке граничных случаев. В отличие от шифрования RSA, для создания подписей существует более простая альтернатива PSS: без ГПСЧ, с единственной хеш-функцией и без дополнения.

Подписи с полнодоменным хешированием

Полнодоменное хеширование (Full Domain Hash, FDH) - простейшая схема подписи, какую только можно представить. Для ее реализации преобразуйте строку байтов `Hash(M)` в число `x` и создайте подпись `y=x^d mod n`, как показано на рисунке 10-4.

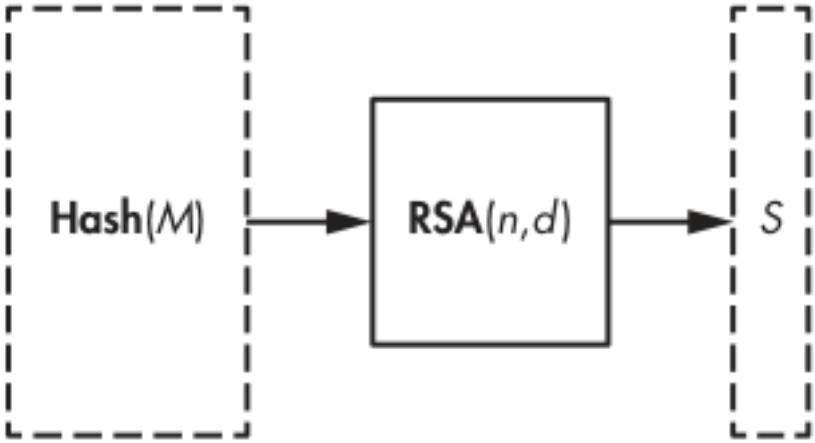

Рисунок 10-4. Подпись сообщения с помощью RSA методом полнодоменного хеширования

Проверка подписи тоже проста. Для подписи, представляющей собой число `y`, вычислите `x=y^e mod n` и сравните результат с `Hash(M)`. Это до скуки просто, детерминированно и вместе с тем безопасно. Так зачем же связываться со сложностью PSS?

PSS появилась позже FDH, в 1996 году, и обладает доказательством безопасности, внушающим больше доверия, чем доказательство FDH. В частности, ее доказательство дает несколько более высокие гарантии безопасности, чем доказательство FDH, а использование случайности помогло усилить это доказательство.

Именно эти более сильные теоретические гарантии служат главной причиной, по которой многие криптографы предпочитают PSS, а не FDH, однако большинство современных приложений с PSS могли бы перейти на FDH без сколько-нибудь значимой потери безопасности. Тем не менее в некоторых ситуациях разумно применять PSS вместо FDH, потому что случайность PSS защищает ее от некоторых атак на реализацию, например атак со сбоями, которые мы обсудим в разделе «Что может пойти не так».

В любом случае подписи RSA (PSS или FDH) применяются все реже. Популярность приобрели подписи на основе эллиптических кривых, такие как ECDSA и EdDSA, не в последнюю очередь потому, что подпись у них вычисляется намного быстрее (хотя проверка подписи зачастую быстрее с RSA).

Реализации RSA

Надеюсь, вам никогда не придется реализовывать RSA с нуля. Если вас об этом попросят, бегите как можно быстрее и усомнитесь в здравомыслии того, кто это предложил. Криптографам и инженерам потребовались десятилетия, чтобы разработать быстрые, достаточно безопасные и в идеале свободные от разрушительных ошибок реализации RSA, поэтому изобретать RSA заново неразумно. Даже при наличии всей доступной документации на выполнение этой пугающей задачи ушли бы месяцы.

Обычно при использовании RSA в программном обеспечении вы будете обращаться к библиотеке или API, предоставляющим необходимые функции для операций RSA. Например, в пакете `crypto` языка Go есть следующая функция (из go.dev):

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte,
    label []byte) (out []byte, err error)
```

Функция `EncryptOAEP()` принимает хеш-функцию, ГПСЧ, открытый ключ, сообщение и метку (необязательный параметр OAEP), а возвращает шифртекст и код ошибки. При вызове `EncryptOAEP()` она вызывает `encrypt()`, чтобы вычислить функцию RSA для дополненных данных, как показано в листинге 10-3.

```
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
}
```

Листинг 10-3. Реализация основной функции шифрования RSA из криптографической библиотеки языка Go

Главная операция здесь - `c.Exp(m, e, pub.N)`: она возводит сообщение `m` в степень `e` по модулю `pub.N` и присваивает результат переменной `c`.

Если вместо готовой библиотечной функции вы решите реализовать RSA самостоятельно, обязательно опирайтесь на существующую библиотеку больших чисел - набор функций и типов, позволяющих определять и выполнять арифметические операции над большими числами длиной в тысячи битов. Например, можно использовать библиотеку арифметики GNU Multiple Precision (GMP) для C или пакет `big` в Go. (Поверьте, самостоятельно реализовывать арифметику больших чисел вам не захочется.)

Даже если при реализации RSA вы всего лишь вызываете библиотечную функцию, убедитесь, что понимаете ее внутреннее устройство, чтобы оценить риск и его соответствие требованиям безопасности приложения.

Алгоритм быстрого возведения в степень

Возведение в степень - это операция возведения `x` в степень `e` при вычислении `x^e mod n`. При работе с большими числами, как в RSA, наивная реализация этой операции может быть чрезвычайно медленной. Как эффективно вычислять степени?

Для наивного вычисления `x^e mod n` требуется `e-1` умножений, что показывает псевдокод в листинге 10-4.

```
expModNaive(x, e, n) {
    y = x
    for i = 1 to e - 1 {
        y = y * x mod n
    }

    return y
}
```

Листинг 10-4. Наивный алгоритм возведения в степень в псевдокоде, возводящий x в степень e по модулю n

Этот алгоритм прост, но крайне неэффективен. Тот же результат можно получить экспоненциально быстрее, возводя промежуточные значения y в квадрат вместо их умножения, пока не будет достигнуто правильное значение. Это семейство методов называется методом возведения в квадрат и умножения, возведением в степень через возведение в квадрат или двоичным возведением в степень.

Допустим, например, требуется вычислить `3^(65 537) mod 36 567 232 109 354 321`. (Число 65 537 служит открытым показателем в большинстве реализаций RSA.) Можно умножить число 3 само на себя 65 536 раз, а можно подойти к задаче, зная, что 65 537 представляется как `2^16+1`, и вместо этого выполнить последовательность возведений в квадрат. По существу, переменной присваивается начальное значение `y=3`, после чего выполняются следующие операции возведения в квадрат (`y^2`):

1. Задать `y=y^2 mod n` (теперь `y=3^2 mod n`).
2. Задать `y=y^2 mod n` (теперь `y=(3^2)^2 mod n=3^4 mod n`).
3. Задать `y=y^2 mod n` (теперь `y=(3^4)^2=3^8 mod n`).
4. Задать `y=y^2 mod n` (теперь `y=(3^8)^2=3^16 mod n`).
5. Задать `y=y^2 mod n` (теперь `y=(3^16)^2=3^32 mod n`).

И так далее, пока `y=3^(65 536)`, для чего выполняется 16 возведений в квадрат.

Чтобы получить окончательный результат, возвращается `3* y mod n=3^(65 537) mod n=26 652 909 283 612 267`. Иными словами, результат вычисляется всего за 17 умножений вместо 65 536 при наивном методе.

В более общем виде метод возведения в квадрат и умножения последовательно просматривает биты показателя степени, вычисляя квадрат для каждого бита показателя, чтобы удвоить значение степени, и умножая на исходное число при каждом встреченном бите со значением 1. В предыдущем примере показатель 65 537 в двоичной записи имеет вид 10000000000000001: значение y возводилось в квадрат для каждого нового бита, а на исходное число 3 умножалось лишь для самого первого и последнего битов.

В листинге 10-5 показан общий псевдокод такого алгоритма для вычисления `x^e mod n`, когда показатель e состоит из битов `e_(m-1)e_(m-2)... e_1e_0`, где `e_0` - младший бит.

```
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
}
```

Листинг 10-5. Быстрый алгоритм возведения в степень в псевдокоде, возводящий x в степень e по модулю n

Алгоритм `expMod()` работает за время `O(m)`, тогда как наивный алгоритм - за время `O(2^m)`, где `m` - битовая длина показателя. Здесь `O()` - обозначение асимптотической сложности, введенное в

главе 9.

Во всех серьезных системах реализован какой-либо вариант этого простейшего метода возведения в квадрат и умножения. Один из таких вариантов - метод скользящего окна, который при выполнении заданной операции умножения рассматривает блоки битов, а не отдельные биты. В качестве примера можно посмотреть функцию `expNN()` языка Go, исходный код которой доступен по адресу go.dev.

Насколько безопасны эти алгоритмы возведения в степень методом квадратов и умножений? К сожалению, приемы, ускоряющие процесс, зачастую одновременно повышают уязвимость перед некоторыми атаками.

Слабость этих алгоритмов вызвана тем, что операции возведения в степень сильно зависят от значения показателя. Операция `if` в листинге 10-5 выбирает разные ветви в зависимости от того, равен бит показателя 0 или 1. Если бит равен 1, итерация цикла `for` выполняется медленнее, чем при 0, и атакующие, наблюдающие за временем выполнения операции RSA, могут использовать эту разницу во времени, чтобы восстановить закрытый показатель. Это называется атакой по времени. При атаках на аппаратуру биты 1 можно отличить от битов 0, отслеживая энергопотребление устройства и наблюдая, на каких итерациях выполняется дополнительное умножение, раскрывающее, какие биты закрытого показателя равны 1.

В зависимости от типа платформы каналом информации может быть не время выполнения. Например, биты 1 в показателе можно отличить от битов 0, измеряя энергопотребление устройства и наблюдая, на каких итерациях выполняется дополнительное умножение, что раскрывает биты закрытого показателя, равные 1. Это атака путем анализа энергопотребления. Немногие криптографические библиотеки с открытым исходным кодом содержат эффективную защиту от атак таких типов.

Малые показатели для более быстрых операций с открытым ключом

Поскольку вычисление RSA по существу является возведением в степень, его производительность зависит от значений участвующих чисел, особенно показателя. Меньшие показатели требуют меньше умножений и потому могут значительно ускорить вычисление степени.

В принципе открытым показателем `e` может быть любое значение от 3 до `phi(n)-1`, если `e` и `phi(n)` взаимно просты. Но на практике встречаются только малые значения `e`, чаще всего `e=65 537`, поскольку важна скорость шифрования и проверки подписей. Например, Microsoft Windows CryptoAPI поддерживает лишь открытые показатели, помещающиеся в 32-битное целое число. Чем больше `e`, тем медленнее вычисляется `x^e mod n`.

В отличие от размера открытого показателя, закрытый показатель `d` неизбежно примерно столь же велик, как `n`, поэтому расшифрование намного медленнее шифрования, а создание подписи намного медленнее ее проверки. Поскольку `d` является секретом, он должен быть непредсказуемым, а значит, его нельзя ограничить малым значением. Например, если `e` фиксировано равным 65 537, соответствующее `d` обычно имеет тот же порядок величины, что и модуль `n`, то есть близко к `2^2048`, если длина `n` составляет 2048 битов.

Как обсуждалось в разделе «Алгоритм быстрого возведения в степень», возведение числа в степень 65 537 требует 17 умножений, тогда как возведение числа в некоторую 2048-битную степень требует порядка 3000 умножений.

Оценить скорость RSA можно с помощью набора инструментов OpenSSL. Например, в листинге 10-6 показаны результаты операций RSA с длиной ключа 512, 1024, 2048 и 4096 битов на MacBook с набором микросхем M2:

```
Doing 1024 bits private rsa sign ops for 10s: 119019 1024 bits private RSA sign ops in 9.
    93s
Doing 1024 bits public rsa verify ops for 10s: 2573979 1024 bits public RSA verify ops in
    9.93s
Doing 2048 bits private rsa sign ops for 10s: 19184 2048 bits private RSA sign ops in 9.
    93s
Doing 2048 bits public rsa verify ops for 10s: 769696 2048 bits public RSA verify ops in
    9.93s
Doing 4096 bits private rsa sign ops for 10s: 3005 4096 bits private RSA sign ops in 9.92s
Doing 4096 bits public rsa verify ops for 10s: 206367 4096 bits public RSA verify ops in
    9.91s

sign    verify   sign/s verify/s
rsa 1024 bits 0.000083s 0.000004s 11985.8 259212.4
rsa 2048 bits 0.000518s 0.000013s  1931.9  77512.2
rsa 4096 bits 0.003301s 0.000048s   302.9  20824.1
--snip--
```

Листинг 10-6. Пример результатов измерения производительности операций RSA с помощью набора инструментов OpenSSL (версия 3.2.0)

Чтобы понять, насколько проверка быстрее создания подписи, вычислим отношение скорости проверки к скорости подписи. Результаты в листинге 10-6 показывают, что у меня отношения скоростей проверки и подписи составили приблизительно 21,6, 40,1 и 68,7 для модулей длиной 1024, 2048 и 4096 битов соответственно. Разрыв растет вместе с размером модуля, потому что число умножений для операций с e остается постоянным относительно размера модуля (например, 17 операций при `e=65 537`), тогда как операции с закрытым ключом при увеличении модуля всегда требуют больше умножений, поскольку соответственно растет `d`.

Если малые показатели настолько хороши, зачем использовать 65 537, а не, скажем, 3? На самом деле при реализации RSA с безопасной схемой, такой как OAEP, PSS или FDH, показатель 3 вполне допустим (и работает быстрее). Однако криптографы избегают этого, поскольку при `e=3` менее безопасные схемы допускают определенные виды математических атак. Число 65 537 достаточно велико, чтобы избежать таких атак на малый показатель, и содержит всего два ненулевых бита, что сокращает время вычисления `x^(65 537)`. Число 65 537 также имеет особое значение для математиков: это четвертое число Ферма, то есть число вида

```
2^((2^n))+1,
```

поскольку оно равно `2^16+1`, где `16=2^4`, однако для большинства инженеров-криптографов это всего лишь несущественная любопытная деталь.

Китайская теорема об остатках

Самый распространенный прием для ускорения расшифрования и создания подписи (то есть вычисления `y^d mod n`) - китайская теорема об остатках (КТО), которая ускоряет RSA примерно в четыре раза.

КТО позволяет ускорить расшифрование, вычисляя два возведения в степень - по модулю `p` и по модулю `q` - вместо одного по модулю `n`. Поскольку `p` и `q` намного меньше `n`, два «малых» возведения в степень выполняются быстрее, чем одно «большое».

Китайская теорема об остатках относится не только к RSA. Это общий арифметический результат, который в простейшей форме утверждает: если `n=n_1n_2n_3...`, где числа `n_i` попарно взаимно просты (то есть `НОД(n_i,n_j)=1` для любых различных `i` и `j`), то значение `x mod n` можно вычислить из значений `x mod n_1`, `x mod n_2`, `x mod n_3`, ... Например, пусть `n=1155`, что является

произведением простых множителей 3*5*7*11, и требуется определить x, удовлетворяющее условиям x mod 3=2, x mod 5=1, x mod 7=6 и x mod 11=8. (Числа 2, 1, 6 и 8 выбраны произвольно.) Чтобы найти x с помощью китайской теоремы об остатках, вычислите сумму P(n_1)+P(n_2)+..., где P(n_i) определяется следующим образом:

```
P(n_i)=(x mod n_i)* n/n_i*(1/(n/n_i) mod n_i) mod n.
```

Заметьте, что второй множитель, n/n_i, равен произведению всех множителей, кроме данного n_i.

Чтобы применить эту формулу к нашему примеру и восстановить x mod 1155, вычислим P(3), P(5), P(7) и P(11), а затем сложим их, получив выражение

```
[
2*385*(1/385 mod 3)+1*231*(1/231 mod 5)+
6*165*(1/165 mod 7)+8*105*(1/105 mod 11)
] mod n.
```

Здесь я просто применил приведенное выше определение P(n_i). (Математика, позволяющая найти каждое число, несложна, но подробно рассматривать ее я не буду.) Затем выражение можно сократить до [770+231+1980+1680] mod n=41. Поскольку для этого примера я выбрал число 41, результат получен верный.

Применение КТО к RSA проще предыдущего примера, потому что у каждого n всего два множителя - p и q. Имея шифртекст y, который требуется расшифровать, вместо вычисления y^d mod n с помощью КТО вычисляют x_p=y^s mod p, где s=d mod (p-1), и x_q=y^t mod q, где t=d mod (q-1). Объединив эти два выражения, вычисляют x следующим образом:

```
x=(x_p* q*(1/q mod p)+x_q* p*(1/p mod q)) mod n.
```

Вот и все. Это быстрее одного возведения в степень даже с приемом возведения в квадрат и умножения, потому что насыщенные умножениями операции выполняются по модулям p и q - числам, вдвое меньшим n, тогда как сложность возведения в степень растет быстрее линейной. При удвоении размера числа стоимость возведения в степень не просто удваивается - она растет быстрее.

> **Примечание.** В последней операции два числа q*(1/q mod p) и p*(1/p mod q) можно вычислить заранее, а значит, чтобы найти x, потребуется выполнить лишь два умножения и одно сложение по модулю n.

К сожалению, у этих методов есть оговорка, касающаяся безопасности.

Что может пойти не так

Еще изящнее самой схемы RSA разнообразие атак, которые работают либо потому, что реализация раскрывает (или ее можно заставить раскрыть) сведения о своем внутреннем состоянии, либо потому, что RSA используется небезопасным образом. В следующих разделах я разберу два классических примера атак этих типов.

Атака Bellcore на RSA-КТО

Атака Bellcore на RSA - одна из важнейших атак в истории RSA. Впервые обнаруженная в 1996 году, она выделялась тем, что использовала уязвимость RSA к инъекциям сбоев - атакам, которые заставляют часть алгоритма работать неправильно и потенциально выдавать неверные результаты. Например, можно временно нарушить работу аппаратных схем или встроенных систем, резко изменив напряжение питания либо направив лазерный импульс на тщательно выбранный участок микросхемы. Затем атакующие способны воспользоваться вызванными сбоями во внутренней работе алгоритма, наблюдая их влияние на результат. Сравнение правильного результата с ошибочным может дать сведения о внутренних значениях алгоритма, включая секретные. Равным образом попытки выяснить, допустим ли результат, могут привести к утечке полезных для атаки сведений.

Атака Bellcore как раз является такой атакой со сбоем. Она действует на схемы подписи RSA, использующие китайскую теорему об остатках и являющиеся детерминированными, - то есть работает против FDH, но не против вероятностной PSS.

Чтобы понять атаку Bellcore, вспомним из предыдущего раздела, что при использовании КТО результат, равный `x^d mod n`, получается следующим вычислением, где `x_p=y^s mod p` и `x_q=y^t mod q`:

```
x=x_p* q*(1/q mod p)+x_q* p*(1/p mod q) mod n.
```

Теперь предположим, что атакующий вызывает сбой при вычислении `x_q`, в результате чего получается некоторое неправильное значение, отличное от настоящего `x_q`. Назовем это неправильное значение `x'_q`, а полученный конечный результат - `x'`. Тогда атакующий может вычесть неправильную подпись `x'` из правильной подписи `x` и факторизовать `n`; получается следующее:

```
x-x'=(x_q-x'_q)* p*(1/p mod q) mod n.
```

Следовательно, значение `x-x'` кратно `p`, то есть `p` является делителем `x-x'`. Поскольку `p` также является делителем `n`, наибольший общий делитель `n` и `x-x'` дает `p`: `НОД(x-x',n)=p`. После этого можно вычислить `q=n/p` и `d`, полностью взломав подписи RSA.

Вариант этой атаки работает, когда известно только подписываемое сообщение, но не правильная подпись. Существует и похожая атака со сбоем на значение модуля, а не на вычисление значений КТО, однако подробно рассматривать ее здесь я не буду.

Общие закрытые показатели или модули

Теперь я покажу, почему ваш открытый ключ не должен иметь тот же модуль `n`, что и чей-либо другой.

Закрытые ключи, принадлежащие разным системам или людям, должны иметь разные закрытые показатели `d`, даже если ключи используют разные модули. Иначе вы могли бы пытаться расшифровать собственным значением `d` сообщения, зашифрованные для других сторон, пока не наткнетесь на сторону с тем же `d`. Аналогично разные пары ключей должны иметь разные значения модуля `n`, даже если у них разные `d`, поскольку `p` и `q` обычно входят в закрытый ключ. Поэтому, если у меня с вами один и тот же `n`, а следовательно, те же `p` и `q`, я могу вычислить ваш закрытый ключ из вашего открытого ключа `e`, используя `p` и `q`.

Представьте, что вам известны собственный закрытый показатель `d_1` и открытый показатель `e_2` другого человека, с которым у вас общий модуль `n`, но неизвестны его множители `p` и `q`. Как вычислить `p` и `q` из своего закрытого показателя `d_1`, чтобы найти закрытый показатель `d_2` другого

человека? Решение немного техническое, но изящное.

Вспомните, что d и e удовлетворяют равенству ed=kphi(n)+1, где phi(n) является секретом и может раскрыть p и q. Вы не знаете k или phi(n), но можете вычислить kphi(n)=ed-1.

Что можно сделать с kphi(n)? Согласно теореме Эйлера, для любого числа a, взаимно простого с n, выполняется a^(phi(n))=1 mod n. Поэтому по модулю n имеем

```
a^(kphi(n))=(a^(phi(n)))^k=1^k=1 mod n.
```

Поскольку kphi(n) - четное число, его можно записать как 2^st для некоторых чисел s и t. Иными словами, равенство a^(kphi(n))=1 mod n можно будет записать в виде x^2=1 mod n для некоторого x, легко вычисляемого из kphi(n). Такое x можно назвать корнем из единицы.

Ключевое наблюдение состоит в том, что x^2=1 mod n эквивалентно утверждению, что значение x^2-1=(x-1)(x+1) делится на n. Иными словами, x-1 или x+1 должно иметь общий множитель с n, благодаря чему можно получить факторизацию n.

В листинге 10-7 приведена реализация этого метода на Python, где ради простоты для нахождения множителей p и q из n и d используются малые 64-битные числа.

```
from math import gcd

n = 36567232109354321
e = 13771927877214701
d = 15417970063428857

kphi = d*e - 1                    # 1
t = kphi

while t % 2 == 0:                 # 2
    t = divmod(t, 2)[0]

a = 2                             # 3
while a < 100:
    k = t                         # 4
    while k < kphi:
        x = pow(a, k, n)
        if x != 1 and x != (n - 1) and pow(x, 2, n) == 1:  # 5
            p = gcd(x - 1, n)     # 6
            break
        k = k*2
    a = a + 2

q = n//p
assert (p*q) == n                 # 7
print('p = ', p)
print('q = ', q)
```

Листинг 10-7. Программа на Python, вычисляющая простые множители p и q из закрытого показателя d

Программа определяет kphi(n) из e и d **1**, находя число t, для которого kphi(n)=2^st при некотором s **2**. Затем она ищет a и k, для которых (a^k)^2=1 mod n **3**, используя t как начальную точку для k **4**. Когда это условие выполняется **5**, решение найдено. После этого программа определяет множитель p **6** и проверяет **7**, что значение pq равно значению n. Затем она выводит полученные значения p и q:

```
p = 2046223079
q = 17870599
```

Программа правильно возвращает два множителя.

Дополнительная литература

RSA заслуживает отдельной книги. Мне пришлось опустить многие важные и интересные темы, такие как атака Даниэля Блайхенбахера через оракул дополнения на предшественника OAEP (стандарт PKCS#1 v1.5) или атака Мангера через оракул дополнения на OAEP; обе атаки по своей идее похожи на атаку через оракул дополнения на блочные шифры из главы 4. Есть также атака Майкла Винера на RSA с малыми закрытыми показателями и атаки методом Копперсмита на RSA с малыми показателями, которые потенциально также используют небезопасное дополнение.

Чтобы познакомиться с результатами исследований атак по сторонним каналам и защиты от них, обратитесь к материалам семинара CHES, проводимого с 1999 года, по адресу ches.iacr.org. Одним из самых полезных источников при написании этой главы была обзорная работа Дэна Боне «Twenty Years of Attacks on the RSA Cryptosystem» («Двадцать лет атак на криптосистему RSA»), в которой рассматриваются и объясняются важнейшие атаки на RSA. Обязательным чтением об атаках по времени является статья Билли Боба Брамли и Дэна Боне «Remote Timing Attacks Are Practical» («Удаленные атаки по времени практичны»), ценная как аналитическим, так и экспериментальным вкладом. Чтобы узнать больше об атаках со сбоями, прочитайте полную версию статьи об атаке Bellcore «On the Importance of Eliminating Errors in Cryptographic Computations» («О важности устранения ошибок в криптографических вычислениях») Дэна Боне, Ричарда ДеМилло и Ричарда Липтона.

Лучший способ узнать, как работают реализации RSA, хотя порой болезненный и раздражающий, - изучить исходный код широко используемых реализаций. Например, посмотрите реализацию RSA и лежащей в ее основе арифметики больших чисел в OpenSSL, NSS (библиотеке, которую использует браузер Mozilla Firefox), Crypto++ или другом популярном программном обеспечении; изучите как реализацию арифметических операций, так и защиту от атак по времени и со сбоями.

Глава 11. Диффи-Хеллман

В ноябре 1976 года исследователи из Стэнфорда Уитфилд Диффи и Мартин Хеллман опубликовали научную статью «New Directions in Cryptography» («Новые направления в криптографии»), навсегда изменившую криптографию. В их статье были представлены понятия шифрования с открытым ключом и подписей, хотя самих таких схем у авторов в действительности не было; они располагали лишь тем, что назвали криптосистемой с открытым ключом, - протоколом, позволяющим двум сторонам установить общий секрет путем обмена сведениями, видимыми подслушивающему. Сейчас он известен как протокол Диффи-Хеллмана (Diffie-Hellman, DH). До появления протокола Диффи-Хеллмана установление общего секрета требовало утомительных процедур, например ручного обмена запечатанными конвертами.

После того как обменивающиеся данными стороны установят общее секретное значение с помощью протокола DH, они могут использовать этот секрет для создания защищенного канала: преобразовать его в один или несколько симметричных ключей и затем применять их для шифрования и аутентификации последующего обмена данными. Поэтому протокол DH и его варианты называются протоколами согласования ключей.

В первой части этой главы вы прочитаете о математических основах протокола Диффи-Хеллмана, включая вычислительные задачи, на которых основано волшебство DH. Затем вы узнаете о разных версиях протокола Диффи-Хеллмана, с помощью которых можно создавать защищенные каналы. Наконец, поскольку схемы Диффи-Хеллмана безопасны лишь при правильном выборе параметров, вы увидите сценарии, в которых Диффи-Хеллман может дать сбой.

> **Примечание.** Диффи и Хеллман получили престижную премию Тьюринга 2015 года за изобретение криптографии с открытым ключом и цифровых подписей, но признания заслуживают и другие. В 1974 году студент факультета информатики Ральф Меркл представил идею криптографии с открытым ключом в виде головоломок Меркла. Примерно тогда же исследователи британского Центра правительственной связи (Government Communications Headquarters, GCHQ), британского аналога АНБ, открыли принципы, лежащие в основе RSA Ривеста-Шамира-Адлемана и согласования ключей Диффи-Хеллмана, хотя этот факт был рассекречен лишь десятилетия спустя.

Функция Диффи-Хеллмана

Чтобы понимать протоколы согласования ключей DH, необходимо понять их основную операцию - функцию DH. Изначально Диффи и Хеллман определили функцию DH для работы с группами, обозначаемыми `Z_p^*` и состоящими из ненулевых целых чисел по модулю простого числа, обычно обозначаемого `p` (см. главу 9). Другим открытым параметром является число-основание, или генератор, `g`.

В функции DH участвуют два закрытых значения, случайно выбранных двумя обменивающимися данными сторонами из группы `Z_p^*`; обозначим их `a` и `b`. С закрытым значением `a` связано открытое значение `A=g^a mod p`, то есть `g`, возведенное в степень `a` по модулю `p`. Значение `A` отправляется другой стороне в сообщении, видимом подслушивающим. Открытое значение, связанное с `b`, равно `B=g^b mod p` и отправляется владельцу `a`. Таким образом, атакующий может узнать `A` и `B`.

DH работает, объединяя любое из открытых значений с другим закрытым значением так, что результат в обоих случаях одинаков: `A^b=(g^a)^b=g^ab` и `B^a=(g^b)^a=g^ba=g^ab`. Полученное

значение g^ab является общим секретом; затем его передают функции формирования ключа (key derivation function, KDF), чтобы сгенерировать один или несколько общих симметричных ключей. KDF - разновидность хеш-функции, возвращающая выглядящую случайной строку, размер которой равен требуемой длине ключа.

Вот и все. Подобно многим великим научным открытиям (гравитации, относительности, квантовым вычислениям или RSA), прием Диффи-Хеллмана в ретроспективе кажется сравнительно простым.

Однако простота Диффи-Хеллмана может быть обманчива. Во-первых, он работает не с любым простым p и не с любым основанием g. Некоторые значения g ограничивают общие секреты g^ab малым подмножеством возможных значений, тогда как следовало бы ожидать примерно столько возможных значений, сколько элементов в Z_p^*, и, следовательно, столько же возможных значений общего секрета. Для наивысшей безопасности надежные параметры DH должны использовать простое p, при котором (p-1)/2 также является простым. Такое безопасное простое гарантирует, что в группе нет малых подгрупп, которые упростили бы взлом DH. При безопасном простом DH может работать с любым элементом Z_p^*, кроме 1 и p-1; в частности, g=2 немного ускоряет вычисления. Но генерация безопасного простого p занимает больше времени, чем генерация полностью случайного простого.

Например, команда dhparam набора инструментов OpenSSL генерирует только безопасные параметры DH, однако дополнительные проверки, встроенные в алгоритм, значительно увеличивают время выполнения, как показывает листинг 11-1.

```
$ time openssl dhparam 2048
Generating DH parameters, 2048 bit long safe prime, generator 2
This is going to take a long time
--snip--
-----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  7.46s user 0.10s system 99% cpu 7.593 total
```

Листинг 11-1. Измерение времени генерации 2048-битных параметров Диффи-Хеллмана с помощью набора инструментов OpenSSL

Генерация параметров DH с помощью набора инструментов OpenSSL заняла около восьми секунд (нередко наблюдается время генерации порядка 30 секунд или даже более одной минуты).

Для сравнения в листинге 11-2 показано, сколько времени в той же системе занимает генерация параметров RSA такого же размера (то есть двух простых чисел p и q, каждое из которых вдвое короче p, использованного для DH).

```
$ time openssl genrsa 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
.....................................................+++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
--snip--
-----END RSA PRIVATE KEY-----
openssl genrsa 2048  0.16s user 0.01s system 95% cpu 0.171 total
```

Листинг 11-2. Генерация 2048-битных параметров RSA с измерением времени выполнения

Генерация параметров DH заняла примерно в 50 раз больше времени, чем генерация параметров RSA с тем же уровнем безопасности, главным образом из-за дополнительного ограничения на простое число, генерируемое для создания параметров DH.

Задачи Диффи-Хеллмана

Безопасность протоколов DH основана на трудности вычислительных задач, особенно задачи дискретного логарифмирования (DLP) из главы 9. Взломать DH можно, восстановив закрытое значение a из его открытого значения g^a, что сводится к решению экземпляра DLP. Однако при использовании DH для вычисления общих секретов нас интересует не только задача дискретного логарифмирования. Важны также две задачи, относящиеся именно к DH.

Вычислительная задача

Вычислительная задача Диффи-Хеллмана (computational Diffie-Hellman, CDH) состоит в вычислении общего секрета g^ab по одним лишь открытым значениям g^a и g^b без знания секретных значений a и b. Цель состоит в том, чтобы подслушивающий, даже перехватив g^a и g^b, не мог определить общий секрет g^ab.

Если вы умеете решать DLP, то сможете решить и CDH: определив a и b по g^a и g^b, можно вычислить g^ab. Иными словами, DLP по меньшей мере столь же трудна, как CDH. Но достоверно неизвестно, является ли CDH по меньшей мере столь же трудной, как DLP, что сделало бы эти задачи равнотрудными. Иными словами, DLP соотносится с CDH так же, как задача факторизации с задачей RSA. (Напомним, что факторизация позволяет решить задачу RSA, но обратное необязательно верно.)

Диффи-Хеллман имеет и другое сходство с RSA: при заданном размере модуля DH обеспечивает примерно тот же уровень безопасности, что и RSA. Например, протокол DH с 2048-битным простым p обеспечивает приблизительно 90-битную безопасность, как и RSA с 2048-битным модулем. Действительно, самый быстрый способ взломать CDH - решить DLP с помощью алгоритма решета числового поля, метода, похожего, но не идентичного общему методу решета числового поля (GNFS), который взламывает RSA посредством факторизации его модуля.

Задача различения

Когда требуется предположение, кажущееся более сильным, чем трудность CDH, в дело вступает задача различения Диффи-Хеллмана (decisional Diffie-Hellman, DDH). Если даны g^a, g^b и значение, равное либо g^ab, либо g^c для некоторого случайного c (каждый из вариантов выбирается с вероятностью 1/2), задача DDH состоит в том, чтобы определить, было ли выбрано g^ab - общий секрет, соответствующий g^a и g^b.

Опора на DDH, а не CDH важна в следующем случае: представьте, что атакующий способен вычислить только первые 32 бита g^ab по 2048-битным значениям g^a и g^b. Хотя CDH остается нерешенной, поскольку 32 битов может быть недостаточно для полного восстановления g^ab, атакующий все же узнает кое-что об общем секрете, а это может позволить ему нарушить безопасность приложения.

Чтобы атакующий не мог узнать об общем секрете g^ab ничего, это значение должно быть неотличимо от случайного элемента группы, подобно тому как схема шифрования безопасна, когда шифртексты неотличимы от случайных строк. Иными словами, атакующий не должен иметь возможности определить, является ли заданное число значением g^ab или g^c для некоторого случайного c, имея g^a и g^b. Предположение различительной задачи Диффи-Хеллмана состоит в том, что ни один атакующий не может эффективно решить DDH.

Если DDH трудна, то трудна и CDH, и узнать что-либо о g^ab невозможно. Поэтому, если вы умеете решать CDH, то сможете решить и DDH: имея тройку (g^a,g^b,x), вы получите g^ab из g^a и g^b и проверите, равен ли результат заданному x.

Суть в том, что DDH принципиально менее трудна, чем CDH (в частности, DDH, в отличие от CDH, нетрудна над Z_p^*), однако трудность DDH является одним из основных и наиболее изученных предположений в криптографии.

Варианты задачи Диффи-Хеллмана

Иногда криптографы разрабатывают новые схемы и доказывают, что взломать их по меньшей мере столь же трудно, как решить некоторую трудную задачу. Но такими трудными задачами не всегда являются CDH или DDH - это могут быть их варианты. Хотелось бы иметь возможность показать, что взлом криптосистемы столь же труден, как решение CDH или DDH, но для сложных криптографических механизмов это не всегда возможно, обычно потому, что такие схемы включают более сложные операции, чем базовые протоколы Диффи-Хеллмана.

Например, в одной задаче типа DH атакующий, имея g^a, пытается вычислить g^(1/a), где 1/a - обратный к a элемент группы (обычно Z_p^* для некоторого простого p). В другой задаче атакующий может пытаться отличить пары (g^a,g^b) от пар (g^a,g^(1/a)) для случайных a и b. Наконец, в двойной задаче Диффи-Хеллмана атакующий, имея g^a, g^b и g^c, пытается вычислить два значения g^ab и g^ac. Иногда такие варианты DH оказываются столь же трудными, как CDH или DDH, а иногда они принципиально легче и потому обеспечивают более низкие гарантии безопасности. В качестве упражнения попробуйте найти связи между трудностью этих задач и трудностью CDH и DDH. (На самом деле двойная задача Диффи-Хеллмана столь же трудна, как CDH, но доказать это непросто!)

Протоколы согласования ключей

Задача Диффи-Хеллмана предназначена для построения безопасных протоколов согласования ключей, которые с помощью общего секрета защищают обмен данными между двумя или несколькими сторонами, взаимодействующими по сети. Стороны преобразуют этот секрет в один или несколько сеансовых ключей - симметричных ключей, которые шифруют и аутентифицируют сведения, передаваемые в течение сеанса.

Прежде чем изучать реальные протоколы DH, следует узнать, что делает протокол согласования ключей безопасным и как работают более простые протоколы. Начнем обсуждение с распространенного протокола согласования ключей, не основанного на DH.

Согласование ключей без DH

Чтобы получить представление о работе протокола согласования ключей и о том, что означает его безопасность, рассмотрим протокол, который сети 4G и 5G используют для установления связи между SIM-картой и оператором связи: аутентифицированное согласование ключей (authenticated key agreement, AKA). Он не использует функцию Диффи-Хеллмана, а опирается только на операции с симметричным ключом. Работа протокола подробно показана на рисунке 11-1.

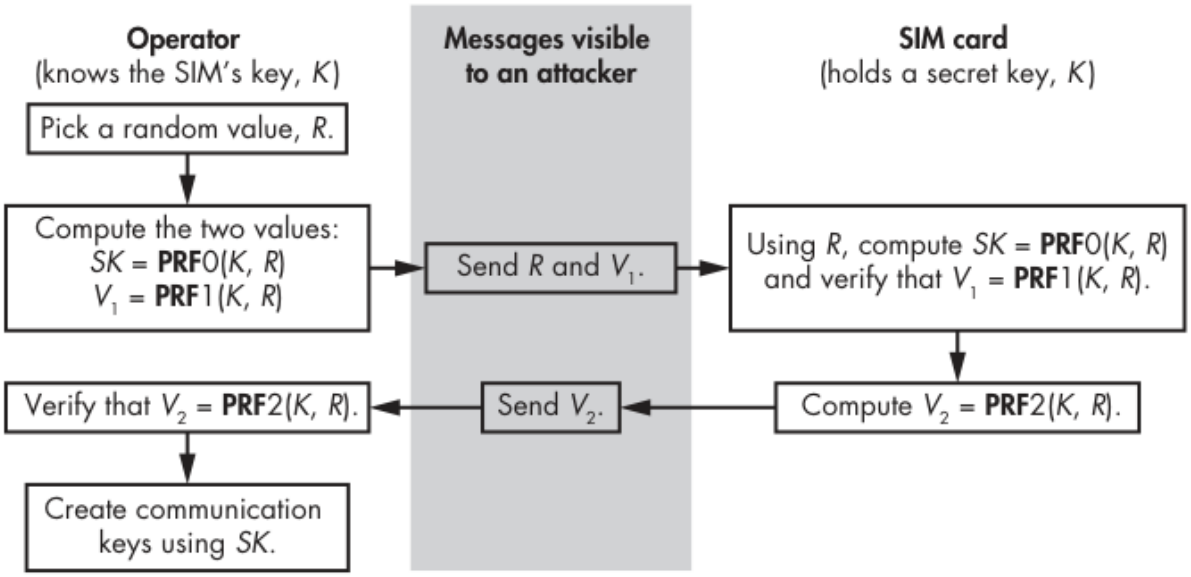

Рисунок 11-1. Протокол AKA в сетях связи 4G и 5G

В этом описании протокола SIM-карта хранит секретный ключ `K`, известный оператору. Оператор начинает сеанс, выбирая случайное значение `R`, а затем вычисляет два значения, `SK` и `V_1`, с помощью двух псевдослучайных функций PRF0 и PRF1. Далее оператор отправляет SIM-карте сообщение со значениями `R` и `V_1`, видимыми атакующим. Получив `R`, SIM-карта располагает всем необходимым для вычисления `SK` с помощью PRF0 и выполняет это вычисление. В итоге обе стороны этого сеанса имеют общий ключ `SK`, который атакующие не могут определить, просто наблюдая за сообщениями, которыми обмениваются стороны, даже если изменяют их или внедряют новые. SIM-карта проверяет, что общается с оператором, заново вычисляя `V_1` с помощью PRF1, `K` и `R`, а затем удостоверяясь, что вычисленное `V_1` совпадает со значением `V_1`, отправленным оператором. После этого SIM-карта вычисляет проверочное значение `V_2` новой функцией PRF2, передав ей на вход `K` и `R`, и отправляет `V_2` оператору. Оператор проверяет, знает ли SIM-карта `K`: вычисляет `V_2` и убеждается, что вычисленное значение совпадает с полученным `V_2`.

В описанном протоколе SIM-карту можно обмануть атакой повторного воспроизведения. По существу, если атакующий перехватит пару `(R,V_1)`, то сможет отправить ее SIM-карте и заставить ту поверить, будто пара пришла от законного оператора, знающего `K`. Чтобы предотвратить эту атаку, протокол включает дополнительные проверки, гарантирующие, что одно и то же `R` не используется повторно.

Проблемы возникают при компрометации `K`. Например, атакующий, скомпрометировавший `K`, может провести атаку «человек посередине» и прослушивать весь обмен открытым текстом. Такой атакующий способен пересылать сообщения между двумя сторонами, выдавая себя одновременно за законного оператора SIM-карты и за саму SIM-карту. Даже если `K` не скомпрометирован во время конкретного сеанса связи, атакующий может записать обмен данными и все сообщения, переданные при согласовании ключей, а позднее, узнав `K`, расшифровать этот обмен с помощью перехваченных значений `R`. Тогда атакующий сможет определить прежние сеансовые ключи и использовать их для расшифрования записанного трафика - в таком случае протокол не обеспечивает прямую секретность.

Модели атак на протоколы согласования ключей

Единого определения безопасности для протоколов согласования ключей нет, и протокол ключа никогда не бывает полностью безопасен вне контекста, без учета модели атаки и целей безопасности. Например, можно утверждать, что предыдущий протокол 4G/5G безопасен, поскольку пассивный атакующий не найдет сеансовые ключи, но он также небезопасен, поскольку утечка ключа K компрометирует весь прошлый и будущий обмен данными.

Для протоколов согласования ключей существуют разные понятия безопасности, а также три основные модели атак, зависящие от сведений, которые раскрывает протокол. От самой слабой к самой сильной это сетевой атакующий, утечка данных и взлом:

Сетевой атакующий. Этот атакующий наблюдает за сообщениями, которыми обмениваются две законные стороны, выполняющие протокол согласования ключей, и может записывать, изменять, отбрасывать или внедрять сообщения. Для защиты от такого атакующего протокол согласования ключей не должен раскрывать никаких сведений об установленном общем секрете.

Утечка данных. В этой модели атакующий получает сеансовый ключ и все временные секреты (например, SK в примере протокола связи) из одного или нескольких выполнений протокола, но не долгосрочные секреты (такие как K в том же протоколе).

Взлом (или компрометация). В этой модели атакующий узнает долгосрочный ключ одной или нескольких сторон. После взлома безопасность более недостижима, поскольку в последующих сеансах протокола атакующий может выдавать себя за одну или обе стороны: это единственная часть информации, идентифицирующая сторону (по крайней мере, в теории, поскольку на практике такие механизмы, как белые списки IP-адресов, могут снизить риск выдачи себя за другую сторону). Тем не менее атакующий не должен иметь возможности восстановить секреты из сеансов, выполненных до получения ключа.

Теперь, рассмотрев модели атак и увидев, на что способен атакующий, изучим цели безопасности, то есть гарантии безопасности, которые должен предоставлять протокол. Протокол согласования ключей можно спроектировать так, чтобы он удовлетворял нескольким целям безопасности. Ниже описаны четыре наиболее значимые из них - от простейшей к наиболее сложной:

Аутентификация. Протокол должен допускать взаимную аутентификацию, при которой каждая сторона может аутентифицировать другую. AKA имеет место, когда протокол аутентифицирует обе стороны.

Управление ключом. Ни одна сторона не должна иметь возможности выбрать окончательный общий секрет или принудительно ограничить его некоторым подмножеством. Рассмотренный выше протокол согласования ключей 4G/5G не обладает этим свойством, поскольку оператор выбирает значение R, полностью определяющее окончательный общий ключ.

Прямая секретность. Даже если все долгосрочные секреты раскрыты, атакующий не должен иметь возможности вычислить общие секреты прежних выполнений протокола, даже если он записал все эти выполнения или мог внедрять либо изменять сообщения в ходе прежних выполнений. Протокол с прямой секретностью, или прямой безопасностью, гарантирует: даже если вам придется передать свои устройства и их секреты некоторому органу власти, он не сможет расшифровать ваш прежний зашифрованный обмен данными. (Протокол согласования ключей 4G/5G прямой секретности не обеспечивает.)

Стойкость к выдаче себя за другую сторону при компрометации ключа (key-compromise impersonation, KCI). KCI происходит, когда атакующий компрометирует долгосрочный ключ одной стороны и может с его помощью выдать себя за другую сторону. Например, протокол согласования ключей 4G/5G допускает тривиальную выдачу себя за другую сторону при компрометации ключа, поскольку обе стороны имеют общий ключ K. В идеале протокол согласования ключей

предотвращает такие атаки.

Производительность

Чтобы быть полезным, протокол согласования ключей должен быть не только безопасным, но и эффективным. При оценке его эффективности следует учитывать несколько факторов, включая число передаваемых сообщений, их длину, вычислительные затраты на реализацию протокола и возможность выполнить предварительные вычисления для экономии времени. Как правило, протокол эффективнее, если передает меньше более коротких сообщений, и лучше всего свести интерактивность к минимуму, чтобы ни одной стороне не приходилось ждать получения сообщения перед отправкой следующего. Обычно эффективность протокола можно измерять его продолжительностью в числе круговых обменов, то есть временем, необходимым, чтобы отправить сообщение и получить ответ.

Время кругового обмена обычно является главной причиной задержки в протоколах, но объем вычислений, выполняемых сторонами, также имеет значение: чем меньше требуется вычислений и чем больше их можно провести заранее, тем лучше. Например, протокол согласования ключей 4G/5G передает два сообщения длиной в несколько сотен битов каждое, причем они должны отправляться в определенном порядке. С этим протоколом можно сэкономить время с помощью предварительных вычислений: оператор может заранее выбрать множество значений `R`, предварительно вычислить соответствующие значения `SK`, `V_1` и `V_2` и сохранить их все в базе данных.

В данном случае предварительные вычисления дают еще одно преимущество - сокращают воздействие на долгосрочный ключ.

Протоколы Диффи-Хеллмана

Функция Диффи-Хеллмана лежит в основе большинства развернутых протоколов согласования с открытым ключом, например в TLS и SSH. Однако единственного протокола Диффи-Хеллмана не существует: есть множество способов использовать функцию DH для установления общего секрета. В следующих разделах мы рассмотрим три протокола. В каждом случае я буду придерживаться обычных условных имен в криптографии: назову две стороны Алисой и Бобом, а атакующую - Евой. Буквой `g` я буду обозначать генератор группы, используемой для арифметических операций, - значение, заранее заданное и известное Алисе, Бобу и Еве.

Анонимный Диффи-Хеллман

Анонимный Диффи-Хеллман - простейший протокол Диффи-Хеллмана. Он анонимен, потому что не аутентифицирован: у участников нет криптографической идентичности, которую могла бы проверить другая сторона, и ни одна сторона не хранит долгосрочного ключа. Алиса не может доказать Бобу, что она Алиса, и наоборот.

В анонимном Диффи-Хеллмане каждая сторона выбирает случайное значение (`a` для Алисы и `b` для Боба), используемое как закрытый ключ, и отправляет другой стороне соответствующий открытый ключ. Подробнее процесс показан на рисунке 11-2.

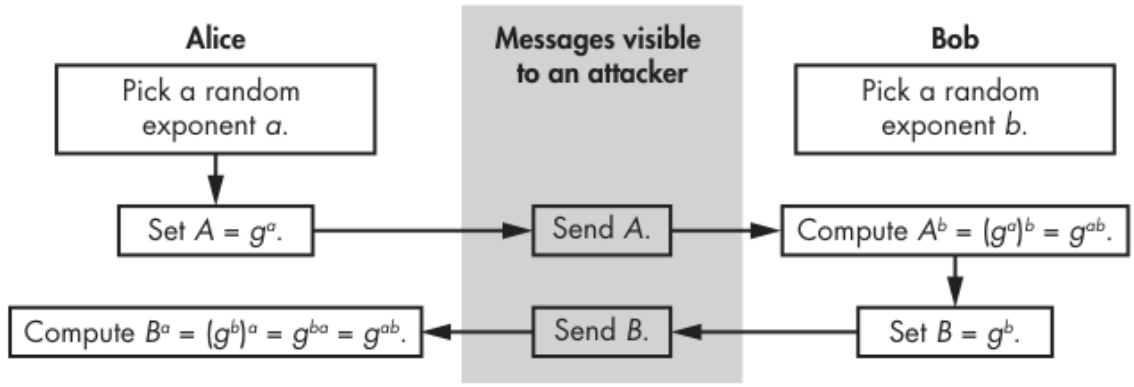

Рисунок 11-2. Анонимный протокол Диффи-Хеллмана

Алиса использует свой показатель `a` и основание группы `g`, чтобы вычислить `A=g^a`, и отправляет его Бобу. Боб получает `A` и вычисляет `A^b`, равное `(g^a)^b`. Теперь Боб получает значение `g^ab` и вычисляет `B` из своего случайного показателя `b` и значения `g`. Затем он отправляет `B` Алисе, и она с его помощью вычисляет `g^ab`. В итоге Алиса и Боб получают одинаковое значение `g^ab`, выполнив похожие операции, в которых и `g`, и полученное значение возводятся в степень закрытого показателя. Простой протокол, безопасный лишь против самых ленивых атакующих.

Атакующий может одолеть анонимный DH атакой «человек посередине». Сетевому атакующему достаточно перехватывать сообщения и выдавать себя за Боба перед Алисой и за Алису перед Бобом, как показано на рисунке 11-3.

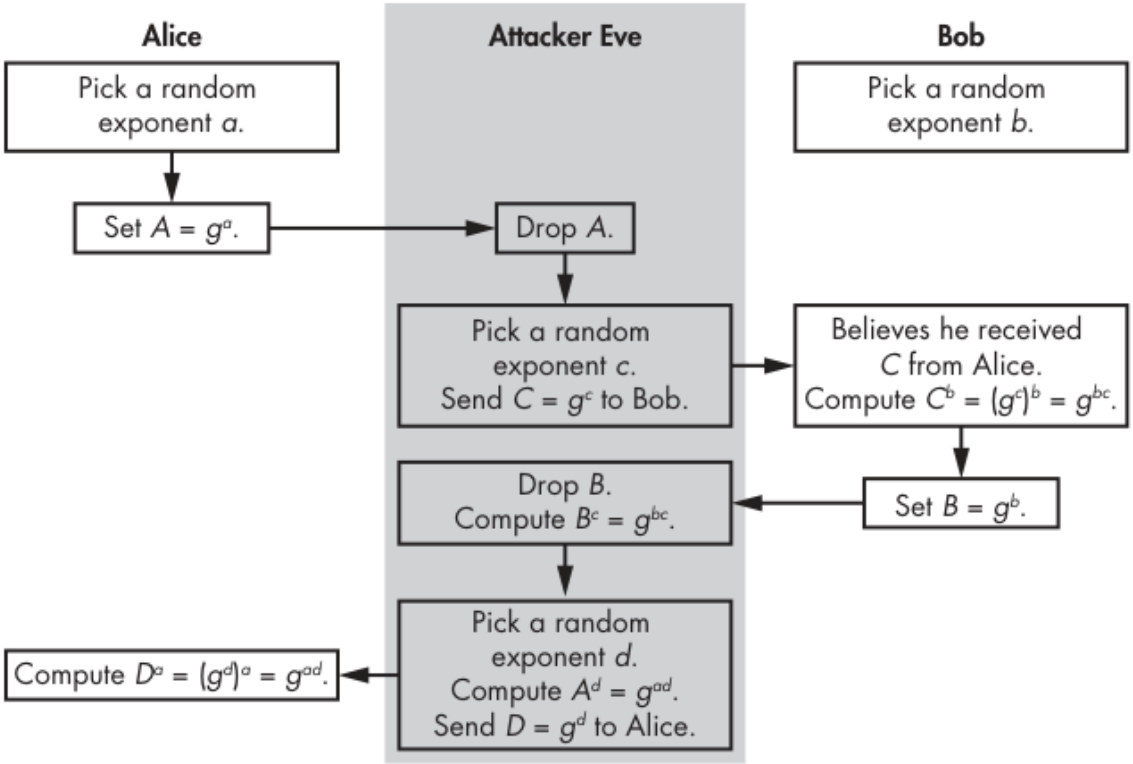

Рисунок 11-3. Атака «человек посередине» на анонимный протокол Диффи-Хеллмана

Как и в предыдущем обмене, Алиса и Боб выбирают случайные показатели `a` и `b`. Теперь Алиса вычисляет и отправляет `A`, но Ева перехватывает и отбрасывает сообщение. Затем Ева выбирает случайный показатель `c`, вычисляет `C=g^c` и отправляет его Бобу. Поскольку в протоколе нет аутентификации, Боб считает, что получил `C` от Алисы, и вычисляет `g^bc`. Затем Боб вычисляет `B` и

отправляет это значение Алисе, однако Ева снова перехватывает и отбрасывает сообщение. Теперь Ева вычисляет `g^bc`, выбирает новый показатель `d`, вычисляет `g^ad`, вычисляет `D` как `g^d` и отправляет `D` Алисе. После этого Алиса также вычисляет `g^ad`.

В результате атаки Ева имеет один общий секрет с Алисой (`g^ad`) и другой общий секрет с Бобом (`g^bc`), тогда как Алиса и Боб считают, что имеют единый общий секрет друг с другом. Завершив выполнение протокола, Алиса формирует из `g^ad` симметричные ключи для шифрования данных, отправляемых Бобу, но Ева перехватывает зашифрованные сообщения, расшифровывает их и заново шифрует для Боба с помощью другого набора ключей, сформированного из `g^bc`, - возможно, предварительно изменив открытый текст. Все это происходит без ведома Алисы и Боба; они обречены.

Чтобы сорвать эту атаку, нужен способ аутентифицировать стороны, благодаря которому Алиса сможет доказать, что она настоящая Алиса, а Боб - что он настоящий Боб. К счастью, такой способ есть.

Аутентифицированный Диффи-Хеллман

Аутентифицированный Диффи-Хеллман устраняет угрозу атак «человек посередине», которым подвержен анонимный DH. В аутентифицированном DH каждая из двух сторон получает и закрытый, и открытый ключ, благодаря чему Алиса и Боб могут подписывать свои сообщения, не позволяя Еве отправлять сообщения от их имени. Здесь подписи вычисляются не функцией DH, а схемой подписи с открытым ключом, например RSA-PSS. В результате, чтобы успешно отправлять сообщения от имени Алисы, атакующему потребуется подделать действительную подпись, что невозможно при безопасной схеме подписи. Работа аутентифицированного DH показана на рисунке 11-4.

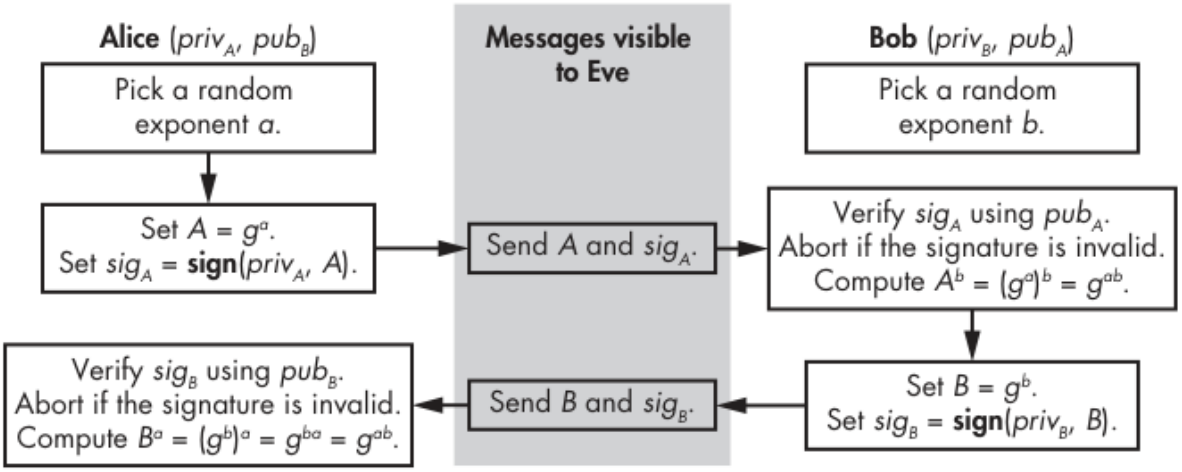

Рисунок 11-4. Аутентифицированный протокол Диффи-Хеллмана

Обозначение Alice (`priv_A`, `pub_B`) в первой строке означает, что Алиса хранит собственный закрытый ключ `priv_A`, а также открытый ключ Боба `pub_B`. Эта пара закрытого и открытого ключей называется долгосрочным ключом, поскольку она задана заранее и остается неизменной при последовательных выполнениях протокола. Алиса может использовать свою пару ключей `priv_A/pub_A` со сторонами помимо Боба, если им известен `pub_A` (откуда он им известен - другой вопрос и одна из сложнейших эксплуатационных проблем в криптографии). Эти долгосрочные закрытые ключи должны храниться в секрете, тогда как открытые ключи считаются известными атакующему.

Алиса и Боб начинают с выбора случайных показателей a и b, как в анонимном DH. Затем Алиса вычисляет A и подпись sig_A сочетанием своей функции подписи sign, своего закрытого ключа priv_A и A. После этого Алиса отправляет A и sig_A Бобу, который проверяет sig_A ее открытым ключом pub_A. Если подпись недействительна, Боб понимает, что сообщение пришло не от Алисы, и отбрасывает A.

Если подпись верна, Боб вычисляет g^ab из A и своего случайного показателя b. Затем он вычисляет B и собственную подпись сочетанием функции sign, своего закрытого ключа priv_B и B. Он отправляет B и sig_B Алисе, которая пытается проверить sig_B открытым ключом Боба pub_B. Алиса вычисляет g^ab только после успешной проверки подписи Боба.

Безопасность против сетевых атакующих

Аутентифицированный DH безопасен против сетевых атакующих, поскольку те, не зная показателей DH, не могут узнать ни одного бита сведений об общем секрете g^ab. Аутентифицированный DH также обеспечивает прямую секретность: даже если в некоторый момент атакующий скомпрометирует одну из сторон, как в рассмотренной ранее модели взлома, он узнает закрытые ключи подписи, но не эфемерные показатели DH, а потому не сможет узнать значение ни одного из прежних общих секретов.

Аутентифицированный вариант DH дает лишь частичную защиту от управления ключом. Алиса не может подобрать особые значения a, чтобы ограничить выбор общего секрета g^ab, поскольку ей еще неизвестно g^b, которое влияет на результат не меньше a. (Исключение возникло бы, если бы Алиса выбрала a=0: тогда g^ab=1 при любом b. Поэтому протокол должен отклонять 0, хотя на практике реализации могут этого не делать.) Однако Боб может перебирать разные значения b, пока не найдет то, которое «его устраивает», например такое, при котором первые 16 битов g^ab равны 1.

Можно лишить Боба власти над значением секрета, если первым сообщением, до отправки Алисой ее g^a, Боб отправит Алисе Hash(g^b). Предлагаю вам самостоятельно проанализировать это изменение и понять, почему оно работает (новое отправляемое сообщение является обязательством относительно открытого ключа Боба).

У аутентифицированного DH есть и другие ограничения. Во-первых, Ева может выдать себя за Алису, записав прежние значения A и sig_A и повторно передав их Бобу. Боб ошибочно считает, что имеет общий секрет с Алисой, хотя Ева не может узнать этот секрет, потому что не знает секрет Алисы a. Следовательно, она не сможет вычислить B^a по отправленному Бобом B.

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 выполнить подтверждение ключа, отправив соответственно Hash(pub_A || pub_B || g^ab) и Hash(pub_B || pub_A || g^ab) для некоторой хеш-функции Hash. Обе стороны могут проверить правильность этих хеш-значений, повторно вычислив результат. Разный порядок открытых ключей pub_A || pub_B и pub_B || pub_A гарантирует, что Алиса и Боб отправят разные значения и атакующий не сможет выдать себя за Алису, скопировав хеш-значение Боба.

Безопасность против утечек данных

Большую тревогу вызывает уязвимость аутентифицированного DH перед атакующими с утечкой данных. В такой атаке атакующий узнает значения эфемерных краткосрочных секретов, а именно показателей `a` и `b`, и использует эти сведения, чтобы выдать себя за одну из обменивающихся данными сторон. Если Ева узнает значение показателя `a` вместе со значением `sig_A`, отправленным Бобу, она сможет начать новое выполнение протокола и выдать себя за Алису, как показано на рисунке 11-5.

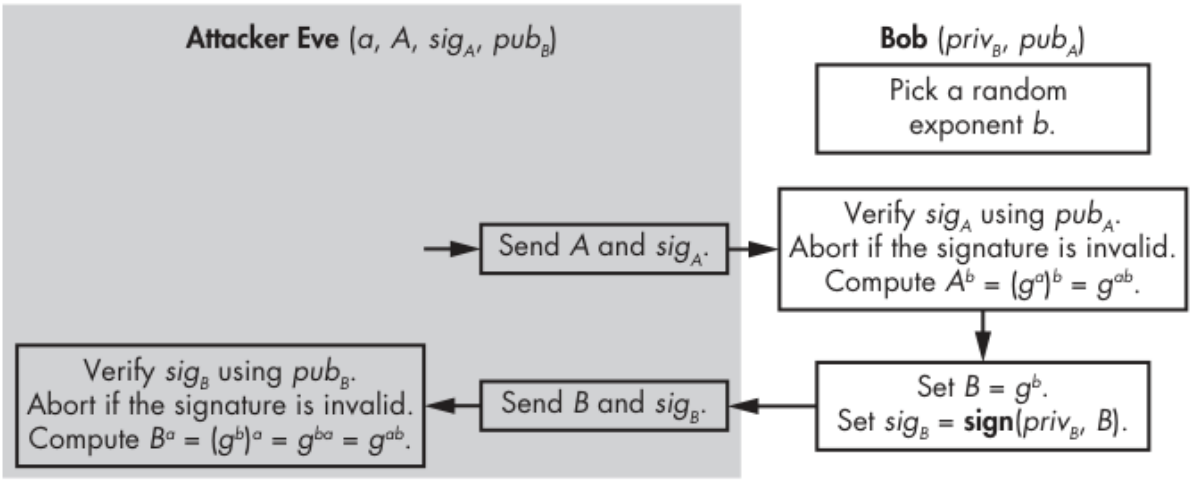

Рисунок 11-5. Атака с выдачей себя за другую сторону на аутентифицированный протокол Диффи-Хеллмана

В этом сценарии атаки Ева узнает значение `a` и повторно передает соответствующее `A` и его подпись `sig_A`, выдавая себя за Алису. Боб проверяет подпись, вычисляет `g^ab` из `A` и отправляет `B` и `sig_B`, после чего Ева с помощью украденного `a` вычисляет `g^ab`. В результате у них появляется общий секрет. Теперь Боб считает, что общается с Алисой.

Защитить аутентифицированный DH от утечки эфемерных секретов можно, включив долгосрочные ключи в вычисление общего секрета, чтобы его нельзя было определить без знания долгосрочного секрета.

Менезес-Кью-Ванстон

Протокол Менезеса-Кью-Ванстона (Menezes-Qu-Vanstone, MQV) - важная веха в истории протоколов на основе DH. Разработанный в 1998 году, MQV был одобрен для защиты самых критичных ресурсов, когда АНБ включило его в Suite B - набор алгоритмов для защиты секретных сведений. (В конце концов АНБ отказалось от MQV, предположительно потому, что он не использовался. Причины этого я вскоре объясню.)

MQV - это Диффи-Хеллман на стероидах. Он безопаснее аутентифицированного DH и превосходит его по характеристикам производительности. В частности, MQV позволяет пользователям отправить всего два сообщения, независимо друг от друга и в произвольном порядке. Кроме того, сообщения могут быть короче, чем в аутентифицированном DH, и не требуется отправлять явные сообщения с подписью или подтверждением. Иными словами, помимо функции Диффи-Хеллмана не нужно применять схему подписи.

Как и в аутентифицированном DH, в MQV Алиса и Боб хранят каждый собственный долгосрочный закрытый ключ, а также долгосрочный открытый ключ другой стороны. Отличие в том, что ключи MQV не являются ключами подписи: они состоят из закрытого показателя x и открытого значения

g^x. Работа протокола MQV показана на рисунке 11-6.

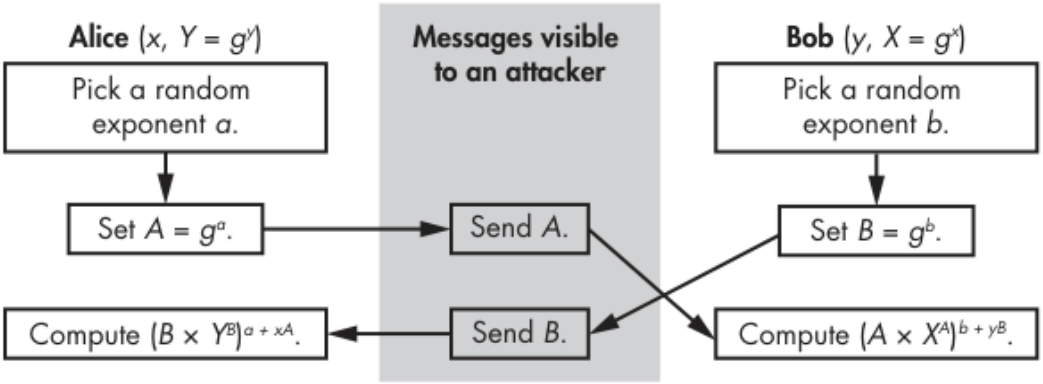

Рисунок 11-6. Протокол MQV

x и y являются долгосрочными закрытыми ключами Алисы и Боба соответственно, а X и Y - их открытые ключи. Боб и Алиса начинают с собственных закрытых ключей и открытого ключа друг друга, представляющего собой g, возведенное в степень закрытого ключа. Каждый выбирает случайный показатель, затем Алиса вычисляет A и отправляет его Бобу, после чего тот вычисляет B и отправляет его Алисе. Получив эфемерный открытый ключ Боба B, Алиса объединяет его со своим долгосрочным закрытым ключом x, своим эфемерным закрытым ключом a и долгосрочным открытым ключом Боба Y, вычисляя (B* Y^B)^(a+xA), как показано на рисунке 11-6. Раскрывая это выражение, получаем

```
(B* Y^B)^(a+xA)
=(g^b*(g^y)^B)^(a+xA)
=(g^(b+yB))^(a+xA)
=g^((b+yB)(a+xA)).
```

Тем временем Боб вычисляет (A* X^A)^(b+yB), и можно убедиться, что оно равно значению, вычисленному Алисой:

```
(A* X^A)^(b+yB)
=(g^a*(g^x)^A)^(b+yB)
=(g^(a+xA))^(b+yB)
=g^((a+xA)(b+yB))
=g^((b+yB)(a+xA)).
```

У Алисы и Боба получается одно и то же значение g^((b+yB)(a+xA)), то есть общий секрет у них одинаков.

В отличие от аутентифицированного DH, MQV нельзя взломать одной лишь утечкой эфемерных секретов. Знание a или b не позволит атакующему определить окончательный общий секрет, поскольку для его вычисления нужны долгосрочные закрытые ключи.

Что происходит в самой сильной модели атаки - модели взлома, когда скомпрометирован долгосрочный ключ? Если Ева скомпрометирует долгосрочный закрытый ключ Алисы x, ранее установленные общие секреты останутся в безопасности, поскольку при их вычислении участвовали также эфемерные закрытые ключи Алисы.

Однако MQV не обеспечивает совершенной прямой секретности из-за следующей атаки. Допустим, Ева перехватывает сообщение Алисы A и заменяет его своим A=g^a для некоторого выбранного Евой a. Тем временем Боб отправляет B Алисе (а Ева записывает значение B) и вычисляет общий ключ. Если позднее Ева скомпрометирует долгосрочный закрытый ключ Алисы x, она сможет определить ключ, вычисленный Бобом в этом сеансе. Это нарушает прямую секретность, поскольку теперь Ева восстановила общий секрет прежнего выполнения протокола. Однако на практике риск можно устранить шагом подтверждения ключа, при котором Алиса и Боб обнаружат, что их ключи

не совпадают, и прервут протокол до формирования каких-либо сеансовых ключей.

Несмотря на изящество и безопасность, MQV редко используется на практике по двум причинам. Раньше он был обременен патентами, что мешало его широкому распространению. Кроме того, правильно реализовать MQV труднее, чем кажется. На деле при сопоставлении с возросшей сложностью преимущества MQV в безопасности часто считаются незначительными по сравнению с более простым аутентифицированным DH.

Что может пойти не так

Протоколы Диффи-Хеллмана могут эффектно давать сбой самыми разными способами. В следующих разделах выделены некоторые случаи, часто наблюдаемые на практике.

Общий секрет не хешируется

Я уже упоминал, что общий секрет, завершающий обмен в сеансе DH (`g^ab` в наших примерах), служит входом для формирования сеансовых ключей, но сам ключом не является. И не должен являться. Симметричный ключ должен выглядеть случайным, и каждый его бит должен с одинаковой вероятностью быть равен 0 или 1. Но `g^ab` не случайная строка; это случайный элемент некоторой математической группы, биты которого могут быть смещены в сторону 0 или 1. Случайный элемент группы отличается от случайной строки битов.

Представьте, например, что вы работаете в мультипликативной группе `Z_13^*={1,2,3,...,12}` и используете `g=2` как генератор группы, то есть `g^i` пробегает все значения `Z_13^*` при `i` от 1 до 12: `g^1=2`, `g^2=4`, `g^3=8`, `g^4=3` и так далее. Если показатель `g` случаен, получится случайный элемент `Z_13^*`, однако кодировка элемента `Z_13^*` в виде 4-битной строки не будет равномерно случайной: вероятность быть 0 или 1 неодинакова для разных битов. В `Z_13^*` у семи значений старший бит равен 0 (числа от 1 до 7 в группе), а у пяти старший бит равен 1 (от 8 до 12). Иными словами, этот бит равен 0 с вероятностью `7/12~=0,58`, тогда как случайный бит в идеале должен быть равен 0 с вероятностью 0,5. Более того, 4-битные последовательности 1101, 1110 и 1111 никогда не появятся.

Чтобы избежать подобных смещений в сеансовых ключах, сформированных из общего секрета DH, используйте криптографическую хеш-функцию, например BLAKE3 или SHA-3, а еще лучше - функцию формирования ключа (KDF). Пример конструкции KDF - HKDF, то есть KDF на основе HMAC (определенная в RFC 5869), однако сегодня BLAKE2 и SHA-3 имеют специальные режимы работы в качестве KDF.

Анонимный Диффи-Хеллман из TLS 1.0

Протокол TLS обеспечивает безопасность защищенных сайтов HTTPS, а также многих других протоколов, например передачи электронной почты с помощью Simple Mail Transfer Protocol (SMTP). TLS принимает несколько параметров, включая тип протокола Диффи-Хеллмана, который будет использоваться. Из соображений обратной совместимости TLS поддерживает анонимный DH начиная с версии 1.0 и заканчивая 1.2 (то есть без какой-либо аутентификации сервера), но не поддерживает его в версии 1.3. Поскольку DH безопасен только против пассивных атакующих, он может создавать ложное ощущение безопасности.

В исходной документации TLS риски этого протокола описаны так (rfc-editor.org):

> Полностью анонимные соединения обеспечивают защиту только от пассивного прослушивания. Если для проверки того, что завершающие сообщения не были заменены атакующим, не используется независимый защищенный от подделки канал, то в средах, где вызывают опасение активные атаки «человек посередине», требуется аутентификация сервера.

Небезопасные параметры группы

В январе 2016 года сопровождающие набора инструментов OpenSSL исправили уязвимость высокой степени опасности (CVE-2016-0701), которая позволяла атакующему использовать небезопасные параметры Диффи-Хеллмана. Первопричина уязвимости состояла в том, что OpenSSL позволял пользователям работать с небезопасными параметрами группы DH (а именно небезопасным простым `p`), вместо того чтобы выдать ошибку и полностью прервать протокол до выполнения любых арифметических операций.

По существу, OpenSSL принимал простое число `p`, мультипликативная группа `Z_p^*` которого (где выполняются все операции DH) содержала малые подгруппы. Как вы узнали в начале этой главы, существование малых подгрупп внутри большей группы в криптографическом протоколе опасно, поскольку ограничивает общие секреты гораздо меньшим множеством возможных значений, чем при использовании всей группы `Z_p^*`. Хуже того, атакующий может подобрать показатель DH `x`, который при объединении с открытым ключом жертвы `g^y` раскрывает сведения о закрытом ключе `y`, а в конечном счете и весь ключ.

Хотя сама уязвимость относится к 2016 году, принцип использованной атаки восходит к статье Чэ Хун Лима и Пил Чун Ли 1997 года «A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup» («Атака с восстановлением ключа на схемы на основе дискретного логарифма с использованием подгруппы простого порядка»). Исправление уязвимости просто: принимая простое `p` в качестве модуля группы, протокол должен проверить, что `p` является безопасным простым, удостоверившись, что `(p-1)/2` также простое. Это гарантирует, что в группе `Z_p^*` не будет малых подгрупп и атака на эту уязвимость не удастся.

Дополнительная литература

Углубить знания о протоколах согласования ключей DH можно, прочитав ряд стандартов и официальных публикаций, включая ANSI X9.42, RFC 2631 и RFC 5114, IEEE 1363 и NIST SP 800-56A. Они служат справочными материалами для обеспечения совместимости и содержат рекомендации по параметрам групп.

Чтобы больше узнать о сложных протоколах DH (таких как MQV и родственные ему HMQV и OAKE, среди прочих) и их понятиях безопасности (включая атаки с неизвестным общим ключом и атаки на представление группы), прочитайте статью Хуго Кравчика 2005 года «HMQV: A High-Performance Secure Diffie-Hellman Protocol» (eprint.iacr.org) и статью Эндрю К. Яо и Юньлэя Чжао 2011 года «A New Family of Implicitly Authenticated Diffie-Hellman Protocols» (eprint.iacr.org). В этих статьях операции Диффи-Хеллмана записываются иначе, чем в этой главе. Например, общий секрет обозначается `xP`, а не `g^x`. Как правило, умножение там заменяется сложением, а возведение в степень - умножением, потому что такие протоколы обычно определены не над группами целых чисел, а над эллиптическими кривыми, с которыми вы познакомитесь в главе 12.

Глава 12. Эллиптические кривые

Появление криптографии на эллиптических кривых (elliptic curve cryptography, ECC) в 1985 году произвело революцию в криптографии с открытым ключом. Она мощнее и эффективнее таких альтернатив, как RSA и классический Диффи-Хеллман: ECC с 256-битным ключом надежнее RSA с 4096-битным ключом. Но она также сложнее.

Подобно RSA, ECC состоит главным образом из умножений больших чисел, однако выполняет их для объединения точек на математической кривой, называемой эллиптической кривой (кстати, к эллипсу она отношения не имеет). Дело осложняется тем, что существует множество типов эллиптических кривых - простых и сложных, эффективных и неэффективных, безопасных и небезопасных в зависимости от сценария применения.

Органы стандартизации начали принимать ECC только в начале 2000-х годов, а в крупных криптографических программах она появилась еще позже: OpenSSL добавила ECC в 2005 году, а средство защищенного подключения OpenSSH - лишь в 2011-м. ECC используется в большинстве соединений HTTPS, в мобильных телефонах и на таких блокчейн-платформах, как Bitcoin и Ethereum. Действительно, эллиптические кривые позволяют выполнять распространенные операции криптографии с открытым ключом

• шифрование, подпись и согласование ключей - быстрее их классических аналогов. Большинство криптографических приложений, основанных на задаче дискретного логарифмирования (DLP), также работают на основе ее аналога для эллиптических кривых, ECDLP, за одним заметным исключением - протоколом Secure Remote Password (SRP).

Эта глава посвящена применениям ECC и рассказывает, когда и почему следует предпочесть ECC алгоритму RSA или классическому Диффи-Хеллману, а также как выбрать подходящую эллиптическую кривую для своего приложения.

Что такое эллиптическая кривая?

Эллиптическая кривая - это кривая на плоскости, множество точек с координатами x и y. Уравнение кривой определяет все принадлежащие ей точки. Например, кривая `y=3` является горизонтальной прямой с вертикальной координатой 3, кривые вида `y=ax+b` при фиксированных числах `a` и `b` - прямыми линиями, `x^2+y^2=1` - окружностью радиуса 1 с центром в начале координат и так далее. Каким бы ни был тип кривой, ее точки - пары `(x,y)`, удовлетворяющие уравнению кривой.

В криптографии эллиптическая кривая обычно имеет уравнение вида `y^2=x^3+ax+b` (форма Вейерштрасса), где константы `a` и `b` определяют форму кривой. Например, на рисунке 12-1 показана эллиптическая кривая, удовлетворяющая уравнению `y^2=x^3-4x`.

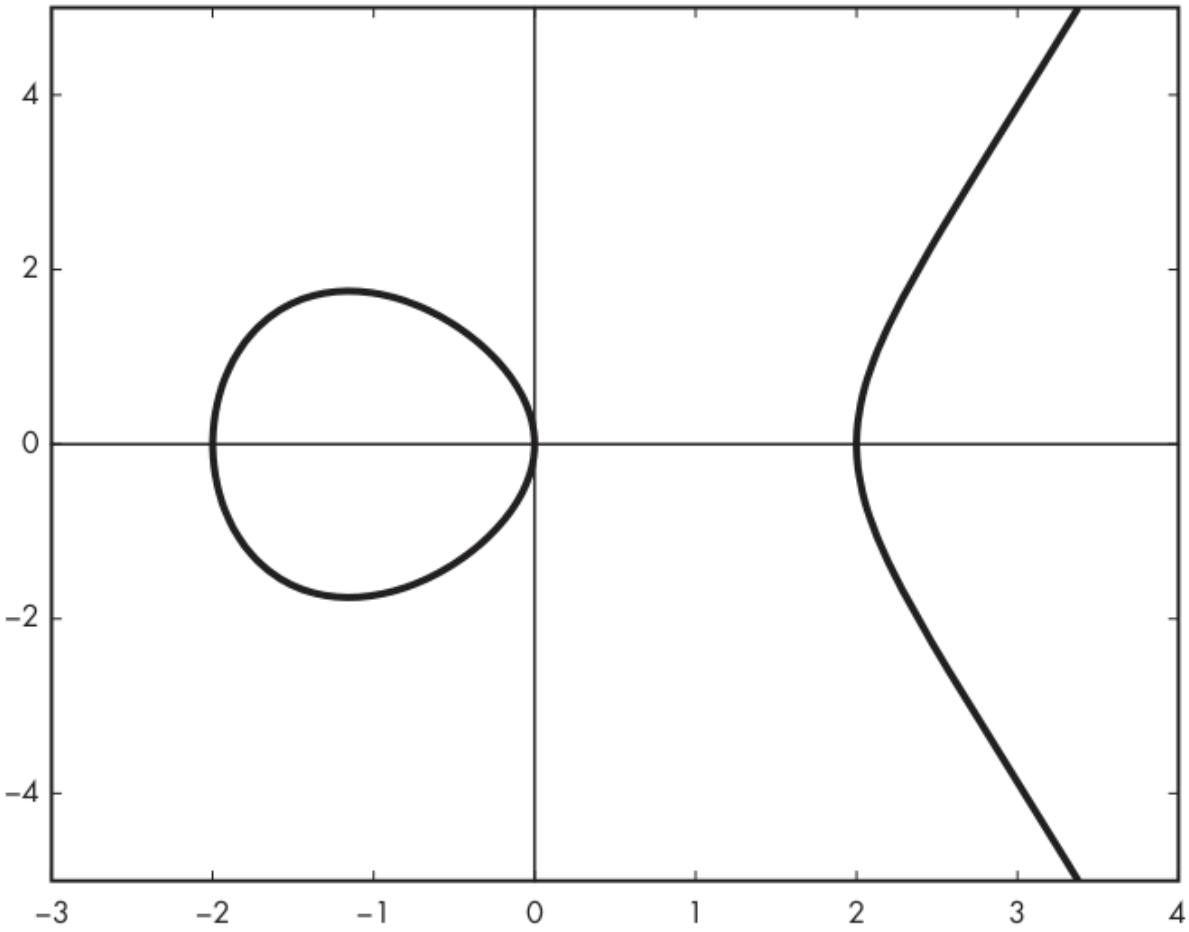

Рисунок 12-1. Эллиптическая кривая с уравнением `y^2=x^3-4x`, показанная над вещественными числами

На рисунке показаны все точки, образующие кривую при `x` от -3 до 4: как точки слева, где кривая похожа на окружность, так и справа, где она похожа на параболу. Координаты `(x,y)` всех точек удовлетворяют уравнению кривой `y^2=x^3-4x`. Например, при `x=0` имеем `y^2=x^3-4x=0^3-4*0=0`; следовательно, `y=0` является решением, и точка `(0,0)` принадлежит кривой. Аналогично при `x=2` решением уравнения является `y=0`, то есть точка `(2,0)` принадлежит кривой.

> **Примечание.** В этой главе я сосредоточусь на простейшем и наиболее распространенном типе эллиптических кривых - кривых с уравнением вида `y^2=x^3+ax+b`, - однако уравнения других эллиптических кривых имеют иную форму. Например, кривые Эдвардса задаются уравнениями вида `x^2+y^2=1+dx^2y^2`. Криптографы иногда используют кривые Эдвардса (например, в схеме Ed25519).

При использовании эллиптических кривых в криптографии крайне важно отличать принадлежащие кривой точки от остальных, поскольку точки вне кривой часто создают угрозу безопасности. Однако уравнение кривой не всегда имеет решения, по крайней мере на плоскости натуральных чисел. Например, чтобы найти точки с горизонтальной координатой `x=1`, решим `y^2=x^3-4x` и получим `y^2=x^3-4x=1^3-4*1`, то есть -3. Но у `y^2=-3` нет решения, потому что не существует числа, квадрат которого равен -3. Решение существует среди комплексных чисел (`sqrt(3)i`), но на практике криптография на эллиптических кривых использует только натуральные числа (ECC работает с целыми числами по модулю простого числа, непосредственно либо через многочлены). Поскольку для `x=1` уравнение кривой не имеет решения, в этой позиции на оси `x` у кривой нет точки, что видно на рисунке 12-1.

Если попытаться решить уравнение для x=-1, получится y^2=-1+4=3 с двумя решениями (y=sqrt(3)~=1,73 и y=-sqrt(3)~=-1,73), то есть квадратным корнем из 3 и его отрицательным значением. Квадрат обоих значений равен 3, и на рисунке 12-1 при x=-1 есть две точки. В более общем случае кривая симметрична относительно оси x для всех удовлетворяющих ее уравнению точек (как и все эллиптические кривые вида y^2=x^3+ax+b, за исключением y=0).

Эллиптические кривые над целыми числами

А теперь небольшой сюрприз: кривые в криптографии на эллиптических кривых на самом деле не похожи на рисунок 12-1. Они не являются ни кривыми, ни эллипсами. Вместо этого они похожи на рисунок 12-2 - облако точек, а не кривую. Что же происходит?

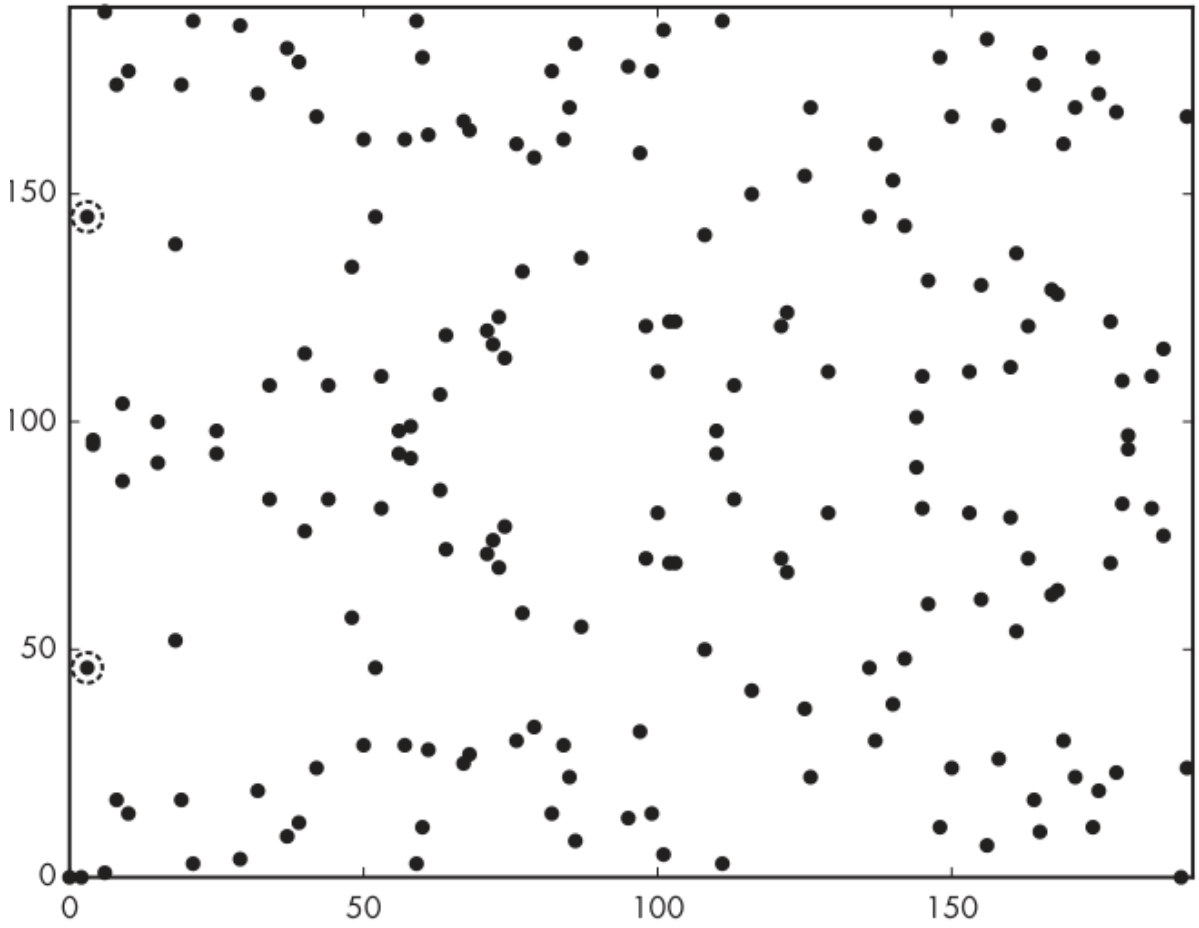

Рисунок 12-2. Эллиптическая кривая с уравнением y^2=x^3-4x над Z_191, множеством целых чисел по модулю 191

Рисунки 12-1 и 12-2 основаны на одном уравнении кривой, y^2=x^3-4x, но показывают ее точки относительно разных множеств чисел. На рисунке 12-1 показаны точки кривой над множеством вещественных чисел, включающим отрицательные числа, десятичные дроби и так далее. Например, непрерывная кривая содержит точки при x=2,0, x=2,1, x=2,00002 и так далее. На рисунке 12-2, напротив, показаны только целые числа, удовлетворяющие уравнению, то есть десятичные дроби исключены. В частности, на рисунке 12-2 показана кривая y^2=x^3-4x относительно целых чисел по модулю 191: 0, 1, 2, 3 и так далее до 190. Это множество чисел обозначается Z_191. (Число 191 здесь ничем не примечательно, кроме того, что оно простое. Я выбрал малое число, чтобы на графике не оказалось слишком много точек.) Поэтому все точки на

рисунке 12-2 имеют координаты x и y, являющиеся целыми числами по модулю 191 и удовлетворяющие уравнению y^2=x^3-4x. Например, при x=2 имеем y^2=0, для чего y=0 является допустимым решением. Следовательно, точка (2,0) принадлежит кривой.

Если x=3, получается уравнение y^2=27-12=15, имеющее два решения в Z_191: 46 и 145. Действительно, 46^2=2116, причем 2116 mod 191=15, а 145^2=21 025, причем 21 025 mod 191=15. Таким образом, обе точки (3,46) и (3,145) принадлежат кривой и показаны на рисунке 12-2 (две точки, обведенные слева).

> **Примечание.** На рисунке 12-2 рассматриваются точки из множества Z_191={0,1,2,...,190}, включающего ноль. Этим оно отличается от групп, обозначаемых Z_p^* (со звездочкой в верхнем индексе), которые мы обсуждали применительно к RSA и Диффи-Хеллману. Причина различия в том, что здесь числа придется как умножать, так и складывать, а потому множество чисел должно включать нейтральный элемент сложения, то есть 0, при котором x+0=x для каждого x в Z_191.
>
> Кроме того, у каждого числа x есть обратный элемент относительно сложения, обозначаемый -x, при котором x+(-x)=0. Например, обратным к 100 в Z_191 является 91, поскольку 100+91 mod 191=0. Такое множество чисел, в котором возможны сложение и умножение и где каждый элемент x имеет обратный относительно сложения (обозначаемый -x), а также обратный относительно умножения (обозначаемый 1/x), кроме нулевого элемента, называется полем. Если поле содержит конечное число элементов, как Z_191 и все поля, используемые в криптографии на эллиптических кривых, оно называется конечным полем.

Закон сложения

Теперь, зная, что точки эллиптической кривой - это координаты (x,y), удовлетворяющие уравнению кривой y^2=x^3+ax+b, посмотрим, как складывать точки эллиптической кривой по закону сложения.

Сложение двух точек

Допустим, требуется сложить две точки эллиптической кривой P и Q, получив новую точку R, которая является суммой этих двух точек. Проще всего понять сложение точек, определив положение R=P+Q на кривой относительно P и Q с помощью геометрического правила: проведите прямую через P и Q, найдите другую точку пересечения этой прямой с кривой, а R будет отражением найденной точки относительно оси x. Например, на рисунке 12-3 прямая через P и Q пересекает кривую в третьей точке между P и Q, а точка P+Q имеет ту же координату x, но противоположную координату y.

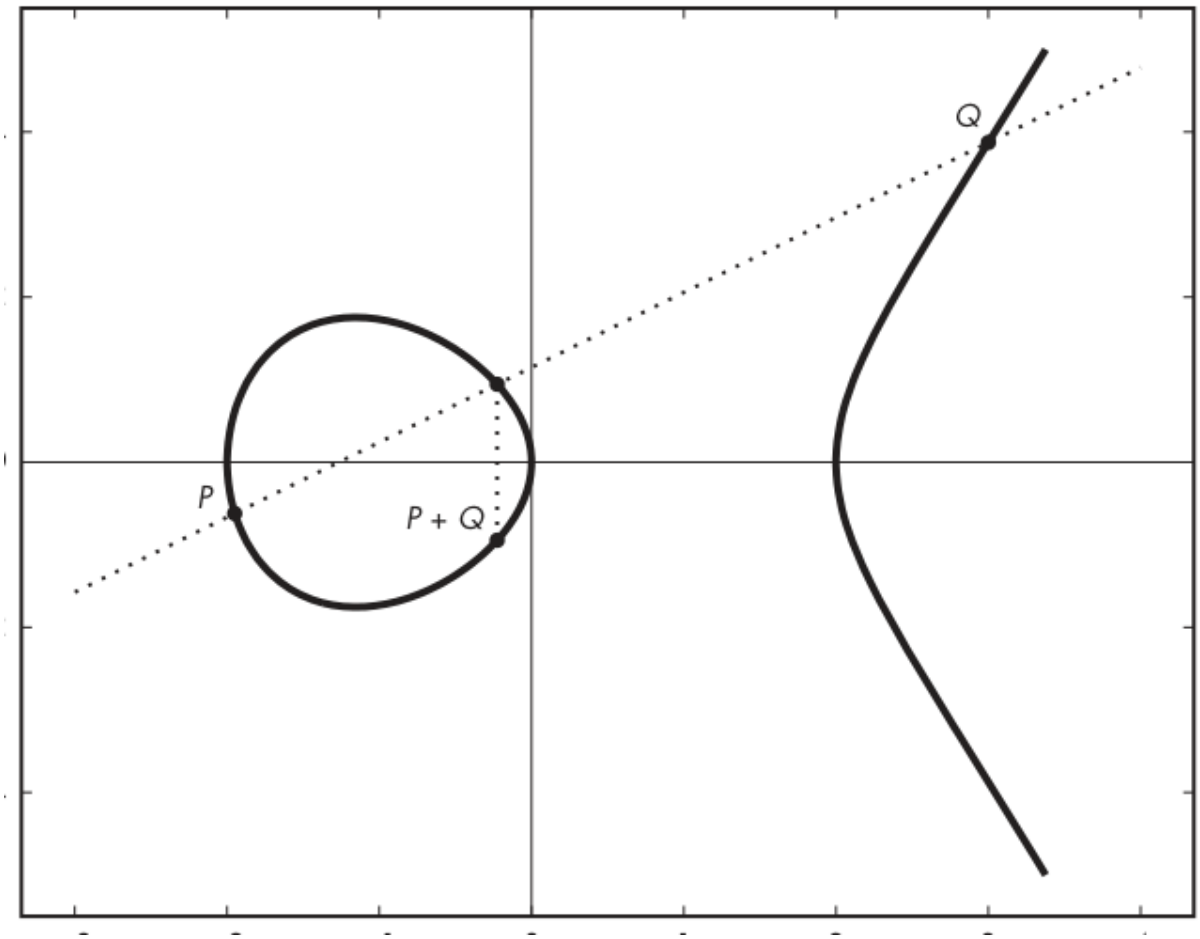

Рисунок 12-3. Общий случай геометрического правила сложения точек на эллиптической кривой

Это геометрическое правило просто, но непосредственно координат точки `R` оно не дает. Координаты `(x_R,y_R)` точки `R` вычисляются по координатам `(x_P,y_P)` точки `P` и координатам `(x_Q,y_Q)` точки `Q` с помощью формул `x_R=m^2-x_P-x_Q` и `y_R=m(x_P-x_R)-y_P`, где `m=(y_Q-y_P)/(x_Q-x_P)` - наклон прямой, соединяющей `P` и `Q`.

К сожалению, эти формулы и прием с проведением прямой на рисунке 12-3 работают не всегда. Например, если `P=Q`, провести прямую между двумя точками нельзя (точка всего одна), а если `Q=-P`, прямая больше не пересекает кривую, поэтому на кривой нет точки, которую можно было бы отразить. Эти случаи мы рассмотрим в следующих разделах.

Сложение точки с противоположной ей точкой

Противоположной для точки `P=(x_P,y_P)` является точка `-P=(x_P,-y_P)`, то есть точка, отраженная относительно оси `x`. Для любой `P` можно записать `P+(-P)=0`, где `0` - точка на бесконечности. Как показано на рисунке 12-4, прямая через `P` и `-P` уходит в бесконечность и никогда не пересекает кривую. Точка на бесконечности играет для эллиптических кривых ту же роль, что ноль для целых чисел, за исключением того, что это всего лишь «виртуальная» точка, которую нельзя расположить в конкретном месте, поскольку она бесконечно далека.

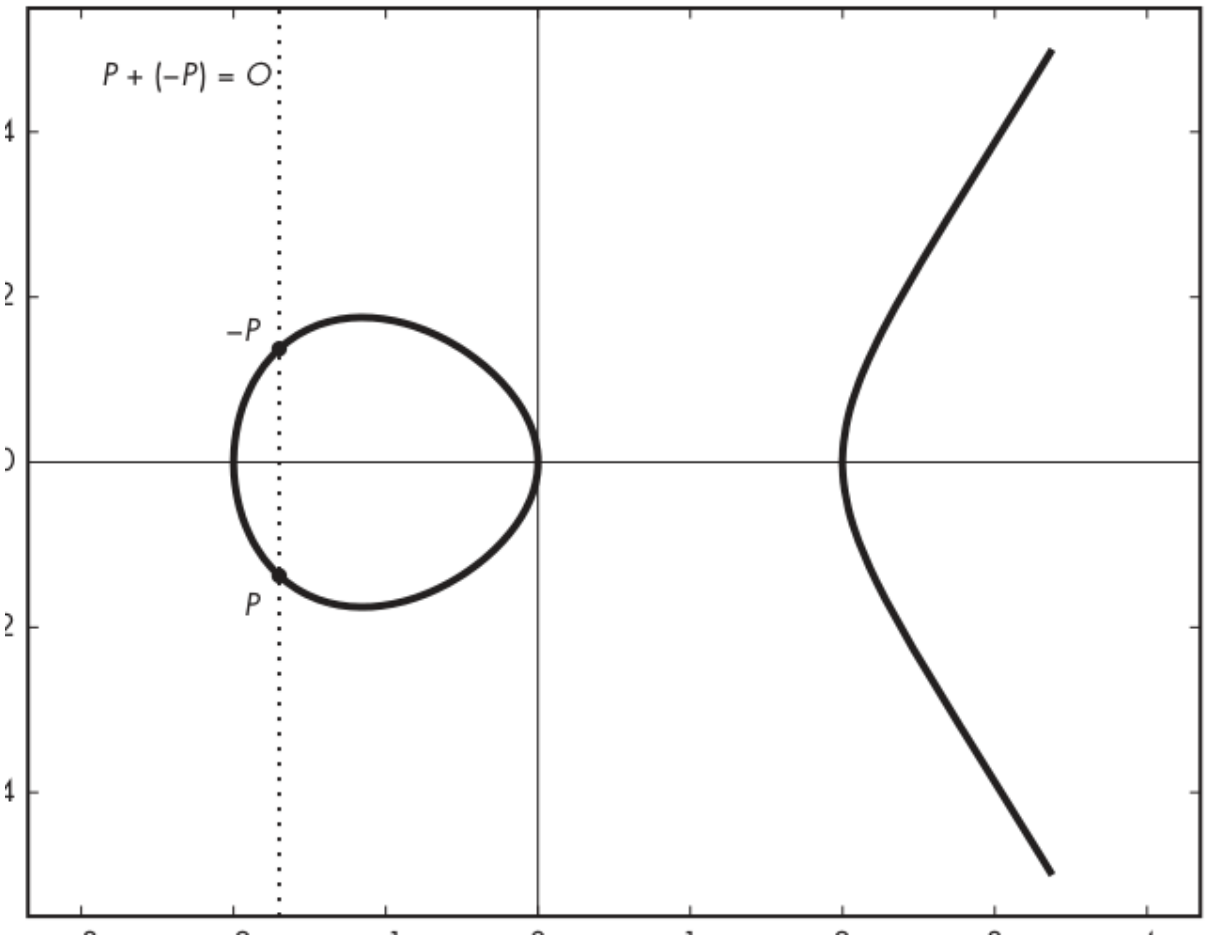

Рисунок 12-4. Геометрическое правило сложения точки с противоположной ей точкой, то есть `P+(-P)=0`, когда прямая через точки больше не пересекает кривую

Удвоение точки

Когда `P=Q` (то есть `P` и `Q` находятся в одной позиции), сложение `P` и `Q` равносильно вычислению `P+P`, или `2P`. Эта операция сложения называется удвоением.

Чтобы найти координаты результата `R=2P`, нельзя использовать геометрическое правило из предыдущего раздела, потому что невозможно провести прямую между `P` и ею самой. Вместо этого проводится касательная к кривой в точке `P`, а `2P` является отрицанием точки, в которой эта прямая пересекает кривую, как показано на рисунке 12-5.

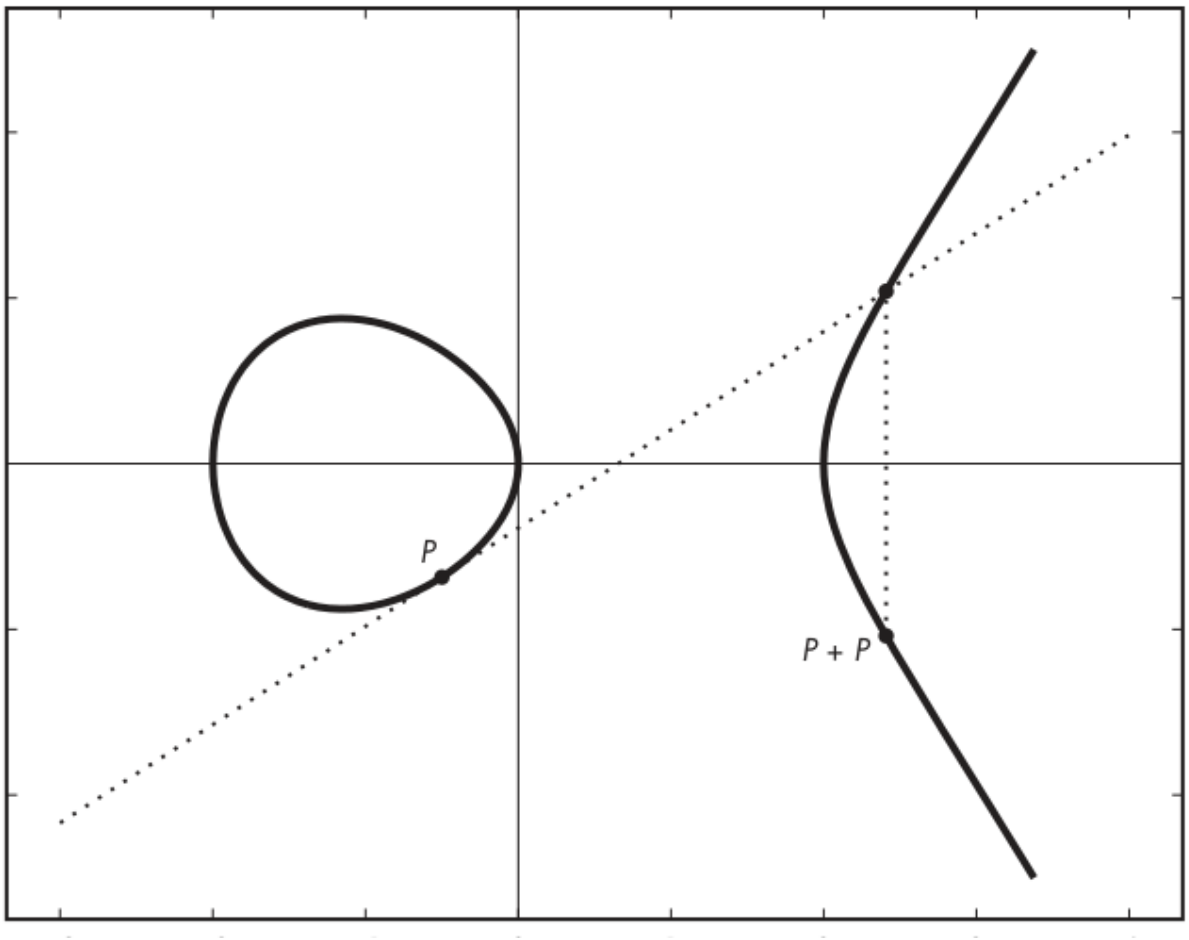

Рисунок 12-5. Геометрическое правило удвоения точки, то есть P+P

Формула для определения координат (x_R,y_R) результата R=P+P немного отличается от формулы для различных P и Q. Основные формулы имеют вид x_R=m^2-x_P-x_Q и y_R=m(x_P-x_R)-y_P, но значение m теперь равно (3x_P^2+a)/(2y_P), где a - параметр кривой, как в y^2=x^3+ax+b.

Умножение точек

Чтобы умножить точку эллиптической кривой на заданное целое число k, точку kP определяют, складывая P с собой k-1 раз. Иными словами, 2P=P+P, 3P=P+P+P и так далее. Чтобы получить координаты x и y точки kP, многократно складывайте P с собой, применяя приведенный выше закон сложения.

Однако для эффективного вычисления kP наивный прием сложения P по закону сложения k-1 раз далек от оптимального. Например, если k велико (порядка, скажем, 2^256), как бывает в криптографических схемах на эллиптических кривых, выполнить k-1 сложений совершенно невозможно.

Экспоненциальное ускорение можно получить, приспособив метод из раздела «Алгоритм быстрого возведения в степень» главы 10, применявшийся для вычисления x^e mod n. Отличие в том, что здесь точки умножаются на целое число, а не вычисляются степени. Но метод можно преобразовать в удвоение и сложение, где умножение заменяется сложением, а возведение в квадрат - удвоением.

Например, чтобы вычислить 8P за три сложения вместо семи при наивном методе, сначала вычислите P_2=P+P, затем P_4=P_2+P_2 и, наконец, P_4+P_4=8P. Это простейший случай, когда

множитель является степенью двойки, например 8=2^3. В противном случае, например для вычисления 10P, действуйте так: заметив, что 10 в двоичной записи имеет вид 1010, начните со старшего («крайнего левого») бита и вычисляйте результат R следующим образом:

Бит 1. Начните, задав R=P.

Бит 0. Выполните удвоение, задав R=2R=2P.

Бит 1. Выполните удвоение и сложение, задав R=2R+P=2(2P)+P=5P.

Бит 0. Выполните удвоение, задав R=2R=2(5P)=10P.

Группы эллиптических кривых

Точки эллиптической кривой можно не только складывать друг с другом, но и использовать для образования структуры группы. Согласно определению группы (см. раздел «Группы» в главе 9), если точки P и Q принадлежат заданной кривой, то P+Q также принадлежит этой кривой.

Кроме того, поскольку сложение ассоциативно, для любых точек P, Q и R выполняется (P+Q)+R=P+(Q+R). В группе точек эллиптической кривой нейтральный элемент называется точкой на бесконечности и обозначается 0, так что P+0=P для любой P. У каждой точки P=(x_P,y_P) есть обратный элемент -P=(x_P,-y_P), при котором P+(-P)=0.

На практике большинство криптосистем на основе эллиптических кривых работает с координатами x и y, являющимися числами по модулю простого числа p (иными словами, числами в конечном поле Z_p). Подобно тому как безопасность RSA зависит от размера используемых чисел, безопасность криптосистемы на эллиптических кривых зависит от числа точек на кривой. Но как узнать число точек эллиптической кривой, то есть ее мощность? Оно зависит от кривой и значения p.

> **Примечание.** Простые поля - не единственные поля, используемые в криптографии на эллиптических кривых. Существуют также двоичные поля, являющиеся расширениями поля из двух элементов (0 и 1), элементы которых представляются многочленами с двоичными коэффициентами. Преимущество арифметики над двоичными полями в том, что ее зачастую проще эффективно реализовать аппаратно в виде логических схем по сравнению с реализацией на универсальном микропроцессоре.

На кривой, определенной над Z_p, имеется порядка p точек, но точное число можно вычислить алгоритмом Шуфа, подсчитывающим точки на эллиптических кривых над конечными полями. Этот алгоритм встроен в SageMath. Например, в листинге 12-1 алгоритм Шуфа используется для подсчета числа точек кривой y^2=x^3-4x над Z_191 с рисунка 12-1.

```
sage: Z = Zmod(191)
sage: E = EllipticCurve(Z, (-4,0))
sage: E.cardinality()
192
```

Листинг 12-1. Вычисление мощности, то есть числа точек на кривой

В листинге 12-1 сначала переменная Z определяется как множество целых чисел по модулю 191; затем переменная E определяется как эллиптическая кривая над Z с коэффициентами -4 и 0. Наконец, вычисляется число точек кривой - ее мощность, порядок группы или просто порядок. При этом в число точек включается точка на бесконечности 0.

Задача ECDLP

В главе 9 была представлена задача дискретного логарифмирования (DLP): найти число y при заданном числе-основании g, если x=g^y mod p для некоторого большого простого числа p. В криптографии на эллиптических кривых есть похожая задача: найти число k при заданной базовой точке P, если точка Q=kP. Это задача дискретного логарифмирования на эллиптической кривой. В задачах на эллиптических кривых вместо чисел используются точки, а вместо возведения в степень - умножение.

Почти вся криптография на эллиптических кривых построена на задаче ECDLP, которая, как и DLP, считается трудной и выдерживает криптоанализ со времени своего появления в 1985 году. Важное отличие от классической DLP в том, что ECDLP позволяет работать с меньшими числами при сопоставимом уровне безопасности.

В общем случае при n-битном параметре p ECC обеспечивает уровень безопасности около n/2 битов. Например, эллиптическая кривая над числами по модулю 256-битного p дает уровень безопасности около 128 битов. Для сравнения: чтобы получить сопоставимый уровень безопасности с DLP или RSA, требуются числа длиной в несколько тысяч битов. Использование меньших чисел в арифметике ECC - одна из причин, по которым она зачастую быстрее RSA или классического Диффи-Хеллмана.

Один из подходов к решению ECDLP - найти коллизию между двумя результатами, c_1P+d_1Q и c_2P+d_2Q. Точки P и Q в этих уравнениях 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-функции в главе 6, коллизия возникает, когда два разных входа дают один и тот же результат. Поэтому для решения ECDLP необходимо найти точки, для которых выполняется

```
c_1P+d_1Q=c_2P+d_2Q.
```

Предположим, четыре коэффициента найдены. Теперь восстановим k. Для этого заменим Q значением kP и получим

```
c_1P+d_1kP =(c_1+d_1k)P,
c_2P+d_2kP =(c_2+d_2k)P.
```

Следовательно, (c_1+d_1k) равно (c_2+d_2k) по модулю числа точек на кривой, которое не является секретом.

Отсюда можно вывести следующее:

```
d_2k-d_1k =c_1-c_2,
k(d_2-d_1) =c_1-c_2,
k =(c_1-c_2)/(d_2-d_1).
```

Итак, найдено k - решение ECDLP. Разумеется, это только общая картина: подробности сложнее и интереснее, особенно способ восстановления коэффициентов c_1, c_2, d_1, d_2.

На практике эллиптические кривые определяются над числами длиной не менее 256 битов, из-за чего атака на криптографию на эллиптических кривых путем поиска коллизии непрактична, поскольку требует до 2^128 операций (стоимость поиска коллизии для 256-битных чисел - см. главу 6).

Согласование ключей Диффи-Хеллмана на эллиптических кривых

Вспомним из главы 11, что в классическом протоколе согласования ключей Диффи-Хеллмана (DH) две стороны устанавливают общий секрет, обмениваясь несекретными значениями. При некотором фиксированном числе `g` Алиса выбирает секретное случайное число `a`, вычисляет `A=g^a` и отправляет `A` Бобу. Затем Боб выбирает секретное случайное `b` и отправляет Алисе `B=g^b`. Оба объединяют свой секретный ключ с открытым ключом другой стороны, получая одно и то же значение `A^b=B^a=g^ab`.

Версия DH для эллиптических кривых, Диффи-Хеллман на эллиптических кривых (elliptic curve Diffie-Hellman, ECDH), идентична классическому DH, но использует другие обозначения. В случае ECC при некоторой фиксированной точке `G` Алиса выбирает секретное случайное число `a`, вычисляет `A=aG` (точку `G`, умноженную на целое число `a`) и отправляет `A` Бобу. Боб выбирает секретное случайное `b`, вычисляет точку `B=bG` и отправляет ее Алисе. Затем оба вычисляют один и тот же общий секрет: `aB=bA=abG`.

ECDH соотносится с задачей ECDLP так же, как DH с DLP: он безопасен, пока ECDLP трудна. Поэтому протоколы DH, опирающиеся на DLP, можно приспособить к работе с эллиптическими кривыми и использовать трудность ECDLP как предположение. Например, аутентифицированный DH и Менезес-Кью-Ванстон (MQV) останутся безопасными при использовании с эллиптическими кривыми. Более того, изначально MQV был определен для работы именно над эллиптическими кривыми.

Подпись с помощью эллиптических кривых

Главный стандартный алгоритм подписи с ECC - алгоритм цифровой подписи на эллиптических кривых (elliptic curve digital signature algorithm, ECDSA). Во многих приложениях он заменил подписи RSA и классические подписи DSA. ECDSA является стандартом NIST, поддерживается протоколами TLS и SSH и служит основным алгоритмом подписи во многих блокчейн-платформах, включая Bitcoin и Ethereum.

Как и всякая схема подписи, ECDSA состоит из алгоритма создания подписи, которым подписывающая сторона создает подпись с помощью закрытого ключа, и алгоритма проверки, которым проверяющая сторона проверяет правильность подписи, имея открытый ключ подписанта. Подписант хранит число `d` в качестве закрытого ключа, а проверяющие стороны хранят открытый ключ `P=dG`. Всем заранее известны используемая эллиптическая кривая, ее порядок (`n`, число точек кривой) и координаты базовой точки `G`.

Создание подписи ECDSA

Чтобы подписать сообщение, подписант сначала хеширует его криптографической хеш-функцией, например SHA-256 или BLAKE2, получая хеш-значение `h`, которое интерпретируется как число от 0 до `n-1`. Затем подписант выбирает случайное число `k` от 1 до `n-1` и вычисляет `kG` - точку с координатами `(x,y)`. Теперь подписант задает `r=x mod n`, вычисляет `s=(h+rd)/k mod n` и использует эти значения как подпись `(r,s)`.

Длина подписи зависит от длины используемых координат. Например, при работе с кривой, координаты которой являются 256-битными числами, и `r`, и `s` имеют длину 256 битов, а вся подпись - 512 битов.

Проверка подписи ECDSA

Алгоритм проверки ECDSA использует открытый ключ подписанта для проверки действительности подписи.

Чтобы проверить подпись ECDSA `(r,s)` и хеш сообщения `h`, проверяющая сторона сначала вычисляет `w=1/s` - обратное значение `s` в подписи, равное `k/(h+rd) mod n`, поскольку `s` определено как `s=(h+rd)/k`. Затем проверяющая сторона умножает `w` на `h`, находя `u` по формуле

```
wh=hk/(h+rd)=u.
```

После этого проверяющая сторона умножает `w` на `r`, находя `v`:

```
wr=rk/(h+rd)=v.
```

Имея `u` и `v`, проверяющая сторона вычисляет точку `Q` по формуле

```
Q=uG+vP.
```

Здесь `P` - открытый ключ подписанта, равный `dG`, и подпись принимается только в том случае, если координата `x` точки `Q` равна значению `r` из подписи.

Этот процесс работает потому, что на последнем шаге точка `Q` вычисляется подстановкой фактического значения `dG` вместо открытого ключа `P`:

```
uG+vdG=(u+vd)G.
```

Подставив вместо `u` и `v` их фактические значения, получим

```
u+vd=hk/(h+rd)+drk/(h+rd)=(hk+drk)/(h+rd)=k(h+rd)/(h+rd)=k.
```

Это означает, что `(u+vd)` равно значению `k`, выбранному при создании подписи, а `uG+vdG` равно точке `kG`. Иными словами, алгоритм проверки успешно вычисляет точку `kG` - ту же точку, которая была вычислена при создании подписи. Проверка завершается, когда проверяющая сторона подтверждает, что координата `x` точки `kG` равна полученному `r`; в противном случае подпись отклоняется как недействительная.

ECDSA и подписи RSA

Хотя некоторые считают криптографию на эллиптических кривых альтернативой RSA для криптографии с открытым ключом, у ECC и RSA не так много общего. RSA используется только для шифрования и подписей, тогда как ECC - семейство алгоритмов, с помощью которых выполняют шифрование, создают подписи, согласовывают ключи и предоставляют сложные криптографические возможности, например шифрование на основе идентификационных данных (разновидность шифрования, в которой применяются ключи, сформированные из личного идентификатора, например адреса электронной почты).

Сравнивая ECDSA с подписями RSA, вспомним: в подписях RSA подписант использует закрытый ключ `d`, чтобы вычислить подпись `y=x^d mod n`, где `x` - подписываемые данные, а `y` - подпись. При проверке открытый ключ `e` используется для подтверждения того, что `y^e mod n` равно `x`, - этот процесс очевидно проще процесса ECDSA.

Проверка RSA зачастую быстрее проверки подписи ECC, поскольку использует малый открытый ключ `e`. Тогда проверка подписи RSA состоит всего лишь в возведении в степень `y^(65 537) mod n`.

У ECC есть два главных преимущества перед RSA: более короткие подписи и скорость подписания. Поскольку ECC работает с более короткими числами, она создает более короткие подписи, чем RSA (длиной в сотни, а не тысячи битов), что явно выгодно, если приходится хранить или

передавать множество подписей. Создание подписи ECDSA также быстрее, чем RSA, поскольку арифметика ECDSA с намного меньшими числами требует меньше вычислительных ресурсов. Например, листинг 12-2 показывает, что ECDSA приблизительно в 200 раз быстрее при создании подписи и примерно столь же быстра, как RSA, при проверке (измерения на процессоре Apple M2). Заметьте, что в этом примере подписи ECDSA также короче подписей RSA: 512 битов (два элемента по 256 битов каждый) вместо 4096 битов.

```
$ openssl speed ecdsap256 rsa4096
                              sign     verify     sign/s     verify/s
rsa 4096 bits            0.003297s  0.000048s      303.3      20933.4
                              sign     verify     sign/s     verify/s
256 bit ecdsa (nistp256)   0.0000s    0.0000s    63709.1      20979.8
```

Листинг 12-2. Сравнение скорости 4096-битных подписей RSA с 256-битными подписями ECDSA

Сравнивать производительность подписей столь разного размера справедливо, поскольку они обеспечивают сопоставимый уровень безопасности. Однако на практике многие системы используют подписи RSA длиной 2048 битов, которые на порядки менее безопасны, чем 256-битная ECDSA. Благодаря меньшему размеру модуля 2048-битная RSA быстрее 256-битной ECDSA при проверке, однако все равно медленнее при создании подписи, как показывает листинг 12-3.

```
$ openssl speed rsa2048
                              sign     verify     sign/s     verify/s
rsa 2048 bits            0.000520s  0.000013s     1923.7      77221.1
```

Листинг 12-3. Скорость 2048-битных подписей RSA

ECDSA следует предпочитать RSA, кроме случаев, когда критически важна проверка подписи, а скорость ее создания не имеет значения, как в ситуации «подписать один раз, проверить много раз» (например, когда исполняемое приложение Windows подписывается один раз, а затем проверяется всеми выполняющими его системами).

EdDSA и Ed25519

ECDSA появилась в начале 1990-х как версия DSA для эллиптических кривых; DSA - стандарт цифровой подписи АНБ, стандартизированный NIST в 1991 году. Однако еще в 1989 году криптограф Клаус-Петер Шнорр предложил так называемую схему подписи Шнорра - алгоритм, более простой и эффективный, чем (EC)DSA, и также подходящий для эллиптических кривых. Но Шнорр подал заявку на патент, ограничивший использование его схемы подписи, что помешало широкому внедрению и стандартизации.

После истечения срока действия патента Шнорра в 2008 году криптографы Дэниел Дж. Бернштейн, Нильс Дюиф, Таня Ланге, Петер Швабе и Бо-Инь Ян на основе идеи Шнорра создали DSA на кривых Эдвардса (Edwards-curve DSA, EdDSA), подпись только для эллиптических кривых со следующими преимуществами:

• проще ECDSA;
• быстрее ECDSA как при создании, так и при проверке подписи;
• детерминирована (тогда как подписи ECDSA и Шнорра использовали при подписании случайное число), что устраняет риски, связанные с несовершенной случайностью.

Посмотрим, как работает EdDSA в общем виде и чем так привлекательна ее конкретная реализация Ed25519. Для простоты я дам упрощенный обзор этих схем. Полные подробности и обоснование схем можно найти в статье 2011 года «High-Speed High-Security Signatures» («Высокоскоростные подписи с высокой безопасностью») и RFC 8032 «Edwards-Curve Digital Signature Algorithm (EdDSA)».

Заметьте, что я использую обозначения, похожие на обозначения в исходной статье; они могут отличаться от применявшихся ранее в этой книге.

Подпись EdDSA

Как и всякая схема подписи, EdDSA требует для подписания закрытый ключ и сообщение, но, в отличие от ECDSA, закрытый ключ представляет собой случайную строку байтов, а не случайное (скалярное) число. Фактический закрытый скаляр получается хешированием строки закрытого ключа. Это дает несколько преимуществ безопасности, в том числе упрощает приложениям выбор случайного ключа: достаточно взять необработанные байты из ГПСЧ, не проверяя, имеет ли ключ правильный формат и не является ли он слабым. Кроме того, это помогает эффективно получить одноразовое значение из ключа и сообщения, заменяя применение отдельного случайного значения для каждой подписи, как в ECDSA.

Подписание 256-битным закрытым ключом `k` сообщения `M` произвольного размера при заданной базовой точке `B` выполняется следующим образом:

1. Вычислить `a || h=Hash(k)` для хеш-функции, создающей 512-битные значения, где первые 256 битов образуют закрытый скаляр `a`, а последние 256 битов - строку `h`.
2. Определить открытый ключ как `A=aB` (на практике он вычисляется заранее).
3. Вычислить «предварительный хеш» сообщения `r=Hash(h || M)` и точку эллиптической кривой `R=rB`, которая является первой частью подписи - точкой подписи.
4. Вычислить число `S=r+Hash(R,A,M)* a` - вторую часть подписи.

Возвращаемая подпись является парой `(R,S)`.

Таким образом, узким местом вычислений служит умножение скаляра на точку `rB` при фиксированной базовой точке `B`. При длинных сообщениях значительной может оказаться и стоимость двукратного хеширования сообщения. Однако, в отличие от ECDSA, вычислять обратный элемент по модулю не требуется.

На практике реализации можно оптимизировать, например не вычислять открытый ключ заново для каждой подписи. Необходимо также задавать значения правильного типа (например, числа, подлежащие приведению по модулю) и кодировать их в надежном и однозначном формате (как точки кривой).

Проверка EdDSA

При заданных подписи `(R,S)`, сообщении `M`, открытом ключе `A` и базовой точке `B` проверка состоит в подтверждении равенства `SB=R+Hash(R,A,M)A`. Заметьте, что `SB=S* B`.

Если заменить `S` его выражением, вычисленным на шаге 4, `SB` равно следующим значениям:

```
(r+Hash(R,A,M)* a)* B
=rB+Hash(R,A,M)* aB.
```

В этом выражении известно, что `rB=R` и `aB=A`. В результате получается ожидаемое значение `R+Hash(R,A,M)* A`.

По сравнению с ECDSA здесь не требуется вычислять обратный элемент по модулю. Как и ECDSA, схема требует двух умножений скаляра на точку (`SB` и `Hash(R,A,M)A`).

Ed25519

Ed25519 - конкретная реализация EdDSA со следующими параметрами:

• скрученная кривая Эдвардса на основе Curve25519, которую мы рассмотрим в следующем разделе;
• SHA-512 в качестве хеш-функции;
• базовая точка `B`, выбранная для оптимизации эффективности.

По состоянию на 2023 год Ed25519, вероятно, является вторым по популярности алгоритмом подписи на эллиптических кривых благодаря преимуществам в производительности и высоким гарантиям безопасности (см. ed25519.cr.yp.to).

OpenSSH, продукты Apple и многие блокчейн-платформы используют Ed25519 для подписи транзакций. В феврале 2023 года Ed25519 была добавлена в качестве стандарта NIST в состав FIPS 186-5 «Digital Signature Standard» («Стандарт цифровой подписи»).

У Ed25519 возникали некоторые проблемы совместимости, которые вы увидите далее в этой главе.

Шифрование с помощью эллиптических кривых

Хотя эллиптические кривые обычно используются для подписи, с их помощью можно и шифровать. Однако на практике это встречается редко из-за ограничений на размер шифруемого открытого текста: в него помещается около 100 битов открытого текста по сравнению почти с 4000 в RSA при том же уровне безопасности.

Шифрование с помощью эллиптических кривых возможно по интегрированной схеме шифрования на эллиптических кривых (elliptic curve integrated encryption scheme, ECIES) - гибридной схеме, объединяющей асимметричную и симметричную криптографию. Она использует операцию Диффи-Хеллмана для получения общего секретного ключа, который защищает данные аутентифицированным шифром.

Имея открытый ключ получателя `P`, ECIES шифрует сообщение `M` следующим образом:

1. Выбрать случайное число `d` и вычислить точку `Q=dG`, где базовая точка `G` является фиксированным параметром. Здесь `(d,Q)` служит эфемерной парой ключей, используемой только для шифрования `M`.
2. Вычислить общий секрет ECDH как `S=dP`.
3. С помощью функции формирования ключа (KDF) сформировать из `S` симметричный ключ `K`.
4. Зашифровать `M` ключом `K` и симметричным аутентифицированным шифром, получив шифртекст `C` и тег аутентификации `T`.

Таким образом, шифртекст ECIES состоит из эфемерного открытого ключа `Q`, за которым следуют `C` и `T`. Расшифрование несложно: получатель вычисляет `S`, умножая `Q` на свой закрытый показатель, затем формирует ключ `K`, расшифровывает `C` и проверяет `T`.

Выбор кривой

К критериям оценки безопасности эллиптической кривой относятся порядок используемой группы (то есть число ее точек), формулы сложения и происхождение параметров.

Существует несколько типов эллиптических кривых, но для криптографических целей они неравноценны. Коэффициенты a и b в уравнении кривой y^2=x^3+ax+b необходимо выбирать внимательно и в соответствии с установленными критериями безопасности;

иначе кривая может оказаться небезопасной. На практике для шифрования используется признанная кривая, но знание того, что делает кривую безопасной, поможет выбрать одну из нескольких доступных и лучше понять связанные с ней риски. Следует помнить следующее:

• Порядок группы не должен быть произведением малых чисел, иначе решить ECDLP становится намного проще.
• В разделе «Закон сложения» вы узнали, что при сложении точек P+Q для случая Q=P требуется особая формула сложения. К сожалению, отдельная обработка этого случая может раскрыть критически важные сведения, если атакующий способен отличить удвоение от сложения различных точек. Некоторые кривые безопасны благодаря единой формуле для любого сложения точек. (Если кривой не требуется особая формула для удвоения, она допускает унифицированный закон сложения.)
• Если создатели кривой не объясняют происхождение a и b, их можно заподозрить в нечестности, поскольку неизвестно, не выбрали ли они более слабые значения, допускающие какую-либо пока неизвестную атаку на криптосистему.

Рассмотрим несколько самых распространенных кривых, особенно используемых для подписей или согласования ключей Диффи-Хеллмана.

> **Примечание.** Дополнительные критерии и подробные сведения о кривых приведены на safecurves.cr.yp.to.

Кривые NIST

В 2000 году NIST стандартизировал несколько кривых в документе FIPS 186 в разделе «Recommended Elliptic Curves for Federal Government Use» («Рекомендуемые эллиптические кривые для использования федеральным правительством»). Пять кривых NIST, называемых простыми кривыми, работают по модулю простого числа (см. раздел «Эллиптические кривые над целыми числами»). Еще десять кривых NIST работают с двоичными многочленами - математическими объектами, позволяющими эффективнее реализовать алгоритмы аппаратно. (Я не буду подробнее рассматривать двоичные многочлены, поскольку с эллиптическими кривыми они используются редко.)

Наиболее распространены простые кривые NIST. Одна из самых распространенных среди них - P-256, работающая над числами по модулю 256-битного числа p=2^256-2^224+2^192+2^96-1. Уравнение P-256 имеет вид y^2=x^3-3x+b, где b - 256-битное число. NIST также предоставляет простые кривые длиной 192, 224, 384 и 521 бит (это не опечатка: именно 521, а не 512).

Кривые NIST иногда критикуют, поскольку происхождение коэффициента b в их уравнениях известно только АНБ, создавшему эти кривые. Согласно предоставленному объяснению, b является результатом хеширования SHA-1 некоторой выглядящей случайной константы. Например, параметр b кривой P-256 происходит от следующей константы: c49d3608 86e70493 6a6678e1 139d26b7 819f7e90. Никто не знает, почему АНБ выбрало именно эту константу, однако большинство специалистов не считает, что происхождение кривой скрывает какую-либо слабость.

Curve25519

Дэниел Дж. Бернштейн представил миру Curve25519 (обычно произносится «curve-twenty-five-five-nineteen», «кривая-двадцать-пять-пять-девятнадцать») в 2006 году. Стремясь повысить производительность, он спроектировал Curve25519 так, чтобы она была быстрее и использовала более короткие ключи, чем стандартные кривые. Но Curve25519 дает и преимущества в безопасности: в отличие от кривых NIST в ней нет подозрительных констант и можно использовать одну унифицированную формулу как для сложения различных точек, так и для удвоения точки.

Форма уравнения Curve25519, `y^2=x^3+486662x^2+x`, немного отличается от других уравнений в этой главе, но все равно принадлежит семейству эллиптических кривых. Необычная форма уравнения позволяет применять особые методы реализации, благодаря которым Curve25519 быстро работает в программном обеспечении.

Curve25519 работает с числами по модулю `2^255-19` - простого числа, максимально близкого к `2^255`. Коэффициент `b`, равный 486662, является наименьшим целым числом, удовлетворяющим заданным Бернштейном критериям безопасности. В совокупности эти особенности делают Curve25519 более заслуживающей доверия, чем кривые NIST с их сомнительными коэффициентами.

Curve25519 используется повсюду, например в WhatsApp, TLS 1.3, OpenSSH и многих других системах. После этого успеха в феврале 2023 года Curve25519 была добавлена в список одобренных NIST кривых в составе документа SP 800-186 «Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters» («Рекомендации для криптографии на основе дискретного логарифма: параметры области эллиптических кривых»).

> **Примечание.** Подробности и обоснование Curve25519 приведены в презентации Дэниела Дж. Бернштейна 2016 года «The First 10 Years of Curve25519» («Первые десять лет Curve25519») по адресу cr.yp.to.

Другие кривые

На момент написания этой книги большинство криптографических приложений использует кривые NIST или Curve25519 (в том числе посредством Ed25519), но применяются и другие устаревшие стандарты, а комитеты стандартизации продвигают новые кривые. К старым национальным стандартам относятся кривые французского агентства ANSSI и немецкие кривые Brainpool - два семейства, которые не поддерживают полные формулы сложения и используют константы неизвестного происхождения.

Некоторые новые кривые эффективнее старых и не вызывают подозрений; они предлагают разные уровни безопасности и различные способы оптимизации эффективности. Среди примеров Curve41417 - вариант Curve25519, работающий с более крупными числами и обеспечивающий более высокий уровень безопасности (около 200 битов); Ed448-Goldilocks - 448-битная кривая, впервые предложенная в 2014 году и определенная в RFC 8032; а также шесть кривых, предложенных Диего Араньей и соавторами в статье «A Note on High-Security General-Purpose Elliptic Curves» («Заметка о высокобезопасных эллиптических кривых общего назначения», см. eprint.iacr.org), хотя эти кривые используются редко. Особенности этих кривых выходят за рамки книги.

Наконец, инициатива Ristretto (ristretto.group) - это метод построения групп точек простого порядка из эллиптических кривых непростого порядка (и потому имеющих подгруппы). Ristretto предлагает безопасный и однозначный способ представления точек эллиптической кривой, устраняя,

например, риски, связанные со структурой Curve25519.

Что может пойти не так

Недостатком эллиптических кривых является их сравнительная сложность. Большее, чем у RSA или классического Диффи-Хеллмана, число параметров означает большую поверхность атаки и больше возможностей для ошибок проектирования и дефектов реализации. Реализации ECC также могут быть уязвимы для атак по сторонним каналам, особенно их арифметика больших чисел, например когда время вычисления зависит от секретных значений.

В следующих разделах я рассмотрю три примера уязвимостей, которые могут возникать при работе с эллиптическими кривыми, даже если реализация безопасна.

ECDSA с плохой случайностью

Создание подписи ECDSA рандомизировано, поскольку при задании `s=(h+rd)/k mod n` используется секретное случайное число `k`. Однако если повторно применить то же `k` для подписи второго сообщения, атакующий сможет объединить полученные значения `s_1=(h_1+rd)/k` и `s_2=(h_2+rd)/k`, получить `s_1-s_2=(h_1-h_2)/k`, а затем `k=(h_1-h_2)/(s_1-s_2)`. Зная `k`, можно легко восстановить закрытый ключ `d` следующим вычислением:

```
(ks_1-h_1)/r=((h_1+rd)-h_1)/r=rd/r=d.
```

В отличие от подписей RSA, где слабый ГПСЧ не позволяет восстановить ключ, использование неслучайных чисел может привести к восстановлению `k` ECDSA, как произошло при атаке на игровую консоль PlayStation 3 в 2010 году, представленной командой fail0verflow на 27-м конгрессе Chaos Communication Congress в Берлине. Исследователи обнаружили, что одно и то же `k` повторно использовалось для подписи разных игр, после чего смогли найти ключ подписи, позволявший подписать любую программу и тем самым разрешить ее выполнение на консоли.

Атаки с недействительной кривой

ECDH можно изящно взломать, если входные точки не проверяются должным образом. Главная причина в том, что в формулах, дающих координаты суммы точек `P+Q`, коэффициент `b` кривой вообще не участвует; они опираются лишь на координаты `P` и `Q` и коэффициент `a` (при удвоении точки). Печальное следствие: складывая две точки, нельзя быть уверенным, что работа ведется на правильной кривой, поскольку в действительности можно складывать точки другой кривой с иным коэффициентом `b`. Это позволяет взломать ECDH по описанному далее сценарию атаки с недействительной кривой.

Допустим, Алиса и Боб выполняют ECDH и согласовали кривую и базовую точку `G`. Боб отправляет Алисе свой открытый ключ `bG`. Алиса вместо открытого ключа `aG` на согласованной кривой намеренно или случайно отправляет точку другой кривой. К сожалению, эта новая кривая слаба и позволяет Алисе выбрать точку `P`, для которой ECDLP решается легко. Она выбирает точку малого порядка, для которой существует сравнительно малое `k`, такое что `kP=0`.

Боб, считая, что располагает законным открытым ключом, вычисляет то, что принимает за общий секрет `bP`, хеширует его и использует полученный ключ для шифрования данных, отправляемых Алисе. Вычисляя `bP`, Боб, сам того не зная, выполняет операции на более слабой кривой. В результате, поскольку `P` выбрана из малой подгруппы внутри большей группы точек, результат `bP`

также принадлежит этой малой подгруппе, что позволяет атакующему эффективно определить общий секрет `bP`, если известен порядок `P`.

Один из способов предотвратить это - подтвердить, что точки `P` и `Q` принадлежат правильной кривой, проверив, что их координаты удовлетворяют уравнению кривой. Это гарантирует возможность работать только на безопасной кривой.

Такая атака с недействительной кривой была обнаружена в 2015 году в некоторых реализациях протокола TLS, использующего ECDH для согласования сеансовых ключей. (Подробности см. в статье Тибора Ягера, Йорга Швенка и Юрая Соморовского «Practical Invalid Curve Attacks on TLS-ECDH» («Практические атаки с недействительной кривой на TLS-ECDH»).)

Несовместимые правила проверки Ed25519

Вы увидели, что Ed25519 - схема подписи, оптимизированная для высокой эффективности и высоких гарантий безопасности. Разработчики Ed25519 подробно описали ее работу в научной статье и опубликовали хорошо написанную эталонную реализацию. Алгоритм был определен как стандарт IETF в RFC 8032. Можно было бы ожидать, что существует лишь одна версия Ed25519 и разные реализации работают совершенно одинаково: для заданного входного значения все они должны вести себя идентично и возвращать один результат.

К сожалению, это не так. Как документировал криптограф Генри де Валенс в статье «It's 255:19AM. Do You Know What Your Validation Criteria Are?» («Сейчас 255:19. Вы знаете свои критерии проверки?», см. hdevalence.ca), разные реализации Ed25519 применяют разные критерии действительности подписи. Действительно, RFC 8032 не описывает критерии проверки полностью, а многие реализации даже не соблюдают его.

У каждой из 15 реализаций в статье де Валенса были собственные критерии проверки. В частности, зачастую различается проверка точек `R` (из подписи) и `A` (открытого ключа), как и проверка равенства `S* B=R+Hash(R,A,M)* A`. Расхождения могут возникать на нескольких уровнях, включая кодирование точек (способ объединения байтов для описания их координат) и проверку принадлежности точек правильной подгруппе.

В результате разные реализации одной схемы могут вести себя по-разному - например, в блокчейн-сети подписи могут приниматься одними узлами и отклоняться другими, что создает проблему для протокола консенсуса. Все подробности объясняются в публикации де Валенса.

Дополнительная литература

Криптография на эллиптических кривых - увлекательная и сложная тема, включающая много математики. Я не рассмотрел такие важные понятия, как порядок точки, кофактор кривой, проективные координаты, точки кручения и методы решения задачи ECDLP. Если вы склонны к математике, сведения об этих и других связанных темах можно найти в книге Анри Коэна и Герхарда Фрея Handbook of Elliptic and Hyperelliptic Curve Cryptography («Справочник по криптографии на эллиптических и гиперэллиптических кривых», Chapman and Hall/CRC, 2005). Иллюстрированное введение с практическими примерами также дает обзор 2013 года «Elliptic Curve Cryptography in Practice» («Криптография на эллиптических кривых на практике») Йоппе Боса, Алекса Халдермана, Нади Хенингер, Джонатана Мура, Михаэля Нерига и Эрика Вастроу (eprint.iacr.org).

Глава 13. TLS

Протокол защиты транспортного уровня (Transport Layer Security, TLS) - рабочая лошадка безопасности интернета. TLS защищает соединения между серверами и клиентами: между сайтом и его посетителями, почтовыми серверами, мобильным приложением и его серверами либо серверами видеоигры и игроками. Без TLS интернет не был бы особенно безопасным.

TLS не зависит от приложения: его можно использовать как для веб-приложений, опирающихся на протокол HTTP, так и для любой системы, в которой клиентскому компьютеру или устройству требуется установить соединение с удаленным сервером. Например, TLS широко используется для связи между машинами в приложениях интернета вещей (IoT), таких как умные холодильники, взаимодействующие с удаленными серверами.

Эта глава дает сокращенный обзор TLS, который с годами становился все сложнее. К сожалению, сложность и раздутость принесли множество уязвимостей, а ошибки, найденные в его перегруженных реализациях, попадали в заголовки новостей: Heartbleed, BEAST, CRIME и POODLE - все это уязвимости, затронувшие миллионы веб-серверов.

> **Примечание.** Иногда TLS называют Secure Sockets Layer (SSL) - так назывался его предшественник.

В 2013 году инженеры начали работать над TLS 1.3. Как вы узнаете из этой главы, TLS 1.3 отбросил ненужные и небезопасные возможности и заменил старые алгоритмы современными на тот момент. В результате протокол стал проще, быстрее и безопаснее.

Прежде чем изучать работу TLS 1.3, рассмотрим задачу, которую призван решать TLS, и причину его существования.

Целевые приложения и требования

TLS - это буква S в адресах сайтов HTTPS, а замок в адресной строке браузера указывает, что страница защищена. Главной причиной создания TLS была потребность в безопасном просмотре сайтов в таких приложениях, как электронная торговля или интернет-банкинг: сайт аутентифицируется, а трафик шифруется для защиты конфиденциальных сведений - личных данных, номеров кредитных карт и учетных данных пользователей.

TLS также помогает защищать общий обмен данными через интернет, устанавливая между клиентом и сервером защищенный канал, который гарантирует конфиденциальность, аутентичность и неизменность передаваемых данных.

Одна из целей безопасности TLS - предотвращение атак «человек посередине», при которых атакующий перехватывает зашифрованный трафик отправляющей стороны, расшифровывает его для получения открытого содержимого, затем повторно шифрует и отправляет принимающей стороне. TLS отражает такие атаки, аутентифицируя серверы (и при необходимости клиентов) с помощью сертификатов и доверенных центров сертификации, как будет рассказано в разделе «Сертификаты и центры сертификации».

Для широкого распространения TLS должен был удовлетворять еще четырем требованиям: эффективности, совместимости, расширяемости и универсальности. Для TLS эффективность означает минимальные потери производительности по сравнению с незашифрованными соединениями. Это выгодно как серверу (сокращает расходы поставщиков услуг на оборудование), так и клиентам (позволяет избежать заметных задержек и сокращения времени работы мобильных

устройств от батареи). Протокол должен был быть совместимым, чтобы работать на любом оборудовании и в любой операционной системе. Он должен был быть расширяемым, чтобы поддерживать дополнительные возможности или алгоритмы. Наконец, он должен был быть универсальным, то есть не привязанным к определенному приложению. В этом он похож на протокол управления передачей (TCP), которому безразлично, какой прикладной протокол используется поверх него.

Набор протоколов TLS

Для защиты взаимодействия клиента с сервером TLS включает несколько версий нескольких протоколов, образующих набор протоколов TLS. TLS не является транспортным протоколом и обычно располагается между транспортным протоколом (TCP) и протоколом прикладного уровня, например HTTP или SMTP, защищая данные, передаваемые по соединению TCP.

TLS также работает поверх транспортного протокола пользовательских датаграмм (UDP), который применяется для передачи «без установления соединения», когда задержка должна быть минимальной, например при потоковой передаче аудио или видео и в сетевых играх. Однако, в отличие от TCP, UDP не гарантирует доставку или правильный порядок пакетов. Поэтому версия TLS для UDP, Datagram Transport Layer Security (DTLS), немного отличается. Подробнее о TCP и UDP см. книгу Чарльза Козьерока The TCP/IP Guide («Руководство по TCP/IP», No Starch Press, 2005).

Семейства протоколов TLS и SSL

История TLS началась в 1995 году, когда Netscape разработала его предка - протокол SSL. SSL был далек от совершенства, и в SSL 2.0 и SSL 3.0 имелись недостатки безопасности. Никогда не следует использовать SSL, всегда используйте TLS; путаницу усиливает то, что TLS нередко называют SSL, в том числе специалисты по безопасности.

Не все версии TLS безопасны. TLS 1.0 (1999) - наименее безопасная версия, хотя она все же безопаснее SSL 3.0. TLS 1.1 (2006) лучше, но включает ряд слабых алгоритмов. TLS 1.2 (2008) еще лучше, однако сложен и обеспечивает высокую безопасность только при правильной настройке. Кроме того, его сложность повышает риск ошибок в реализациях и неправильных конфигураций. Например, TLS 1.2 поддерживает AES в режиме CBC, который зачастую уязвим для атак через оракул дополнения.

TLS 1.2 унаследовал от ранних версий TLS десятки возможностей и проектных решений, делающих его неоптимальным с точки зрения безопасности и производительности. Чтобы навести порядок, инженеры-криптографы создали TLS заново: сохранили только удачные части и добавили средства безопасности. Результатом стал TLS 1.3 - капитальная переработка, которая упростила раздутую конструкцию и сделала ее безопаснее, эффективнее и проще. По существу, TLS 1.3 - зрелый TLS.

TLS в двух словах

В TLS есть два основных протокола: протокол рукопожатия (или просто рукопожатие) определяет секретные ключи, общие для двух сторон; протокол записей описывает использование этих ключей для защиты данных. Пакеты данных, которые обрабатывает TLS, называются записями. TLS определяет формат пакета для инкапсуляции данных протоколов более высокого уровня с целью передачи другой стороне.

Рукопожатие начинается с того, что клиент инициирует защищенное соединение с сервером. Клиент отправляет начальное сообщение ClientHello с параметрами, в том числе с желаемым шифром. Сервер проверяет сообщение и его параметры, а затем отвечает сообщением ServerHello. Обработав сообщения друг друга, клиент и сервер готовы обмениваться зашифрованными данными с помощью сеансовых ключей, установленных протоколом рукопожатия, как будет показано в разделе «Протокол рукопожатия TLS».

Сертификаты и центры сертификации

Самый важный шаг рукопожатия TLS и основа безопасности TLS - проверка сертификата, при которой сервер использует сертификат для собственной аутентификации перед клиентом.

Сертификат по существу представляет собой открытый ключ, сопровождаемый подписью этого ключа и связанных сведений (включая доменное имя). Например, при подключении к google.com браузер получает сертификат от некоторого сетевого узла, а затем проверяет подпись сертификата, в которой сказано примерно следующее: «Я - google.com, а мой открытый ключ - [ключ]». Если подпись подтверждена, сертификат и его открытый ключ считаются доверенными, и браузер продолжает устанавливать соединение. (Подробности о подписях см. в главах 10 и 12.)

Браузер знает открытый ключ, необходимый для проверки подписи, благодаря центру сертификации (certificate authority, CA), который по существу является открытым ключом, жестко заданным в браузере или операционной системе. Закрытый ключ этого открытого ключа (то есть возможность подписания) принадлежит доверенной организации, которая гарантирует, что открытые ключи в выпущенных ею сертификатах принадлежат заявленным сайтам или субъектам. Иными словами, CA выступает доверенной третьей стороной. Без CA невозможно проверить, принадлежит ли открытый ключ, предоставленный google.com, компании Google, а не подслушивающему, проводящему атаку «человек посередине».

Например, в листинге 13-1 показано, что происходит при использовании инструмента командной строки OpenSSL для установления TLS-соединения с `www.google.com` через порт 443 - сетевой порт, используемый для соединений HTTP на основе TLS (HTTPS).

```
$ openssl s_client -connect www.google.com:443
CONNECTED(00000003)
---
Certificate chain
 0 s:CN = www.google.com
   i:C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
 1 s:C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
   i:C = US, O = Google Trust Services LLC, CN = GTS Root R1
 2 s:C = US, O = Google Trust Services LLC, CN = GTS Root R1
   i:C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIEiDCCA3CgAwIBAgIQNJvIv7ypW9IQHaA7P69MzzANBgkqhkiG9w0BAQsFADBG
MQswCQYDVQQGEwJVUzEiMCAGA1UEChMZR29vZ2xlIFRydXN0IFNlcnZpY2VzIExM
QzETMBEGA1UEAxMKR1RTIENBIDFDMzAeFw0yMzA5MDQwODIzMjlaFw0yMzExMjcw
ODIzMjhaMBkxFzAVBgNVBAMTDnd3dy5nb29nbGUuY29tMFkwEwYHKoZIzj0CAQYI
KoZIzj0DAQcDQgAENEMvWAY0TRTb0w5ZxbUbX/Z+EcviE50SzQzvP/xyyVIaURM4
A0Jer9IJO/6Iq6o2AfDXUxrdBKpSzlzaeFCaqqOCAmgwggJkMA4GA1UdDwEB/wQE
AwIHgDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMBAf8EAjAAMB0GA1UdDgQW
BBRnwQnVC8ok1e6YPIbQwjB+XFpPRjAfBgNVHSMEGDAWgBSKdH+vhc3ulc09nNDi
RhTzcTUdJzBqBggrBgEFBQcBAQReMFwwJwYIKwYBBQUHMAGGG2h0dHA6Ly9vY3Nw
LnBraS5nb29nL2d0czFjMzAxBggrBgEFBQcwAoYlaHR0cDovL3BraS5nb29nL3Jl
cG8vY2VydHMvZ3RzMWMzLmRlcjAZBgNVHREEEjAQgg53d3cuZ29vZ2xlLmNvbTAh
BgNVHSAEGjAYMAgGBmeBDAECATAMBgorBgEEAdZ5AgUDMDwGA1UdHwQ1MDMwMaAv
```

```
oC2GK2h0dHA6Ly9jcmxzLnBraS5nb29nL2d0czFjMy9mVkp4YlYtS3Rtay5jcmww
ggEFBgorBgEEAdZ5AgQCBIH2BIHzAPEAdgDoPtDaPvUGNTLnVyi8iWvJA9PL0RFr
7Otp4Xd9bQa9bgAAAYpfgRjOAAAEAwBHMEUCID+BcS984SEh2E2UrKfLvF2fG7qa
SYkzbELytrDz91wmAiEAlwOPMM26CynmadsqomPMXKdRNMvzdyciHVimh0snrBAA
dwB6MoxU2LcttiDqOOBSHumEFnAyE4VNO9IrwTpXo1LrUgAAAYpfgRkAAAAEAwBI
MEYCIQDZf1tULkVCXRc68zJwgp5WFJUbTxFjz6CP+eLb3dgz3gIhAJ9uS7psu2Gl
HdTXokXTetMY7MCdIcuj60Qm/qTn+1dFMA0GCSqGSIb3DQEBCwUAA4IBAQCIEn0v
QzaqNCOhiI5TKcRhaR24yKid3F57a/GOM1LDE/v7oCm+3fxtvuK9HVa/Dmnvavqp
ci7TpMDj/ocXjE4dL4/yHaVx6GhTDKMW/bbBkDaqXoSdb9lAcUZPLRTV4AjFdjmB
8wZTf95bnfeuKNXWlbo/k/9pRRhFNKKMUI54xLiVdj4wk1EUAsrMTTn+Ol2ZbFeS
s614abBT5W0hhFkLvjvEht8p3UKQwwyhRjZMBsae/d0QfT8hglVtVhhGd7f1hFqI
XSERl8EsyDc3urCsa+RjUjCvE9Q+y2X8WVV0HPCjwsANU56qEZmh4kqgg5paW/SL
maM/8Vny/nKNd6Dj
-----END CERTIFICATE-----
subject=CN = www.google.com

issuer=C = US, O = Google Trust Services LLC, CN = GTS CA 1C3

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 4295 bytes and written 396 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Листинг 13-1. Установление TLS-соединения с `www.google.com` и получение сертификатов для аутентификации соединения

Данные сертификата находятся между маркерами `BEGIN CERTIFICATE` и `END CERTIFICATE`. Перед ними раздел `Certificate chain` содержит описание цепочки сертификатов: строки, начинающиеся с `s:`, описывают субъект сертифицированной стороны, а строки, начинающиеся с `i:`, - издателя подписи. Сертификат 0 получен от `www.google.com`, сертификат 1 принадлежит стороне, подписавшей сертификат 0, а сертификат 2 - стороне, подписавшей сертификат 1.

Сертификат 0 выпустила организация Google (через подразделение Google Trust Services, GTS), которая разрешила выпуск сертификата 0 для доменного имени `www.google.com`, подписав сертификат закрытым ключом GTS CA 1C3. Сертификатом, подтверждающим принадлежность этого ключа иерархии ключей Google, является сертификат 1, подписанный ключом GTS Root R1 - корневым сертификатом Google. Сертификат 2, выпущенный GlobalSign, (признанным центром сертификации), подтверждает, что ключ GTS Root R1 принадлежит организации Google.

В этом примере в операционной системе обычно уже имеются сертификаты 1 и 2, которые она считает доверенными. Тогда достаточно проверить две подписи: подразделения Google GTS CA 1C3 в сертификате 0 и подразделения Google GTS Root R1 в сертификате 1. Если система еще не считает сертификат 2 доверенным, но содержит корневой сертификат GlobalSign (GlobalSign Root CA), потребуется также проверить подпись GlobalSign в сертификате 2.

Организации центров сертификации, такие как Google и GlobalSign, должны заслуживать доверия и выпускать сертификаты только для заслуживающих доверия сторон, а также защищать свои закрытые ключи, чтобы атакующий не мог выпускать сертификаты от их имени (например, чтобы выдать себя за законный сервер `www.google.com`).

Чтобы увидеть содержимое сертификата, введите команду `openssl x509 -text -noout` в терминале Unix, а затем вставьте сертификат из листинга 13-1. Результат показан в листинге 13-2.

```
$ openssl x509 -text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      34:9b:c8:bf:bc:a9:5b:d2:10:1d:a0:3b:3f:af:4c:cf
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = US, O = Google Trust Services LLC, CN = GTS CA 1C3
    Validity
      Not Before: Sep  4 08:23:29 ...
      Not After : Nov 27 08:23:28 ...
    Subject: CN = www.google.com
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
        Public-Key: (256 bit)
        pub:
          04:34:43:2f:58:06:34:4d:14:db:d3:0e:59:c5:b5:
          1b:5f:f6:7e:11:cb:e2:13:9d:12:cd:0c:ef:3f:fc:
          72:c9:52:1a:51:13:38:03:42:5e:af:d2:09:3b:fe:
          88:ab:aa:36:01:f0:d7:53:1a:dd:04:aa:52:ce:5c:
          da:78:50:9a:aa
        ASN1 OID: prime256v1
        NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Extended Key Usage:
        TLS Web Server Authentication
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Subject Key Identifier:
        67:C1:09:D5:0B:CA:24:D5:EE:98:3C:86:D0:C2:30:7E:5C:5A:4F:46
      X509v3 Authority Key Identifier:
        8A:74:7F:AF:85:CD:EE:95:CD:3D:9C:D0:E2:46:14:F3:71:35:1D:27
```

```
Authority Information Access:
        OCSP - URI:http://ocsp.pki.goog/gts1c3
        CA Issuers - URI:http://pki.goog/repo/certs/gts1c3.der
      X509v3 Subject Alternative Name:
        DNS:www.google.com
      X509v3 Certificate Policies:
        Policy: 2.23.140.1.2.1
        Policy: 1.3.6.1.4.1.11129.2.5.3
      X509v3 CRL Distribution Points:
        Full Name:
          URI:http://crls.pki.goog/gts1c3/fVJxbV-Ktmk.crl
      CT Precertificate SCTs:
        Signed Certificate Timestamp:
          Version   : v1 (0x0)
          Log ID    : E8:3E:D0:DA:3E:F5:06:35:32:E7:57:28:BC:89:6B:C9:
                      03:D3:CB:D1:11:6B:EC:EB:69:E1:77:7D:6D:06:BD:6E
          Timestamp : Sep  4 09:23:30.638 2023 GMT
          Extensions: none
          Signature : ecdsa-with-SHA256
            30:45:02:20:3F:81:71:2F:7C:E1:21:21:D8:4D:94:AC:
            A7:CB:BC:5D:9F:1B:BA:9A:49:89:33:6C:42:F2:B6:B0:
            F3:F7:5C:26:02:21:00:97:03:8F:30:CD:BA:0B:29:E6:
            69:DB:2A:A2:63:CC:5C:A7:51:34:CB:F3:77:27:22:1D:
            58:A6:87:4B:27:AC:10
        Signed Certificate Timestamp:
          Version   : v1 (0x0)
          Log ID    : 7A:32:8C:54:D8:B7:2D:B6:20:EA:38:E0:52:1E:E9:84:
                      16:70:32:13:85:4D:3B:D2:2B:C1:3A:57:A3:52:EB:52
          Timestamp : Sep  4 09:23:30.688 2023 GMT
          Extensions: none
          Signature : ecdsa-with-SHA256
            30:46:02:21:00:D9:7F:5B:54:2E:45:42:5D:17:3A:F3:
            32:70:82:9E:56:14:95:1B:4F:11:63:CF:A0:8F:F9:E2:
            DB:DD:D8:33:DE:02:21:00:9F:6E:4B:BA:6C:BB:61:A5:
            1D:D4:D7:A2:45:D3:7A:D3:18:EC:C0:9D:21:CB:A3:EB:
            44:26:FE:A4:E7:FB:57:45
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
      88:12:7d:2f:43:36:aa:34:23:a1:88:8e:53:29:c4:61:69:1d:
      b8:c8:a8:9d:dc:5e:7b:6b:f1:8e:33:52:c3:13:fb:fb:a0:29:
      be:dd:fc:6d:be:e2:bd:1d:56:bf:0e:69:ef:6a:fa:a9:72:2e:
      d3:a4:c0:e3:fe:87:17:8c:4e:1d:2f:8f:f2:1d:a5:71:e8:68:
      53:0c:a3:16:fd:b6:c1:90:36:aa:5e:84:9d:6f:d9:40:71:46:
      4f:2d:14:d5:e0:08:c5:76:39:81:f3:06:53:7f:de:5b:9d:f7:
      ae:28:d5:d6:95:ba:3f:93:ff:69:45:18:45:34:a2:8c:50:8e:
      78:c4:b8:95:76:3e:30:93:51:14:02:ca:cc:4d:39:fe:3a:5d:
      99:6c:57:92:b3:ad:78:69:b0:53:e5:6d:21:84:59:0b:be:3b:
      c4:86:df:29:dd:42:90:c3:0c:a1:46:36:4c:06:c6:9e:fd:dd:
      10:7d:3f:21:82:55:6d:56:18:46:77:b7:f5:84:5a:88:5d:21:
      11:97:c1:2c:c8:37:37:ba:b0:ac:6b:e4:63:52:30:af:13:d4:
      3e:cb:65:fc:59:55:74:1c:f0:a3:c2:c0:0d:53:9e:aa:11:99:
      a1:e2:4a:a0:83:9a:5a:5b:f4:8b:99:a3:3f:f1:59:f2:fe:72:
      8d:77:a0:e3
```

Листинг 13-2. Декодирование сертификата, полученного от `www.google.com`

В листинге показано, как команда `openssl x509` декодирует сертификат, первоначально представленный блоком данных в кодировке base64. Поскольку OpenSSL знает структуру этих данных, он может показать содержимое сертификата, включая серийный номер и сведения о версии, идентификационные сведения, сроки действия (строки `Not Before` и `Not After`), открытый ключ (здесь в виде модуля RSA и его открытого показателя) и подпись перечисленных сведений.

Хотя специалисты по безопасности и криптографы часто утверждают, что вся система сертификатов изначально порочна, она остается одним из лучших доступных решений наряду, например, с принятой в SSH политикой доверия при первом использовании (trust on first use, TOFU).

Протокол записей

Все данные в сеансах связи TLS 1.3 передаются как последовательности записей TLS - пакетов данных, используемых TLS. Протокол записей TLS (уровень записей) по существу является транспортным протоколом, которому безразличен смысл переносимых данных; благодаря этому TLS подходит для любого приложения.

Сначала протокол записей TLS переносит данные, которыми обмениваются при рукопожатии. После завершения рукопожатия, когда обе стороны имеют общий секретный ключ, данные приложения разбиваются на фрагменты, передаваемые внутри записей TLS.

Структура записи TLS

Запись TLS - фрагмент данных размером не более 16 Кбайт со следующей структурой:

• Первый байт представляет тип передаваемых данных и имеет значение 22 для данных рукопожатия, 23 для зашифрованных данных и 21 для предупреждений. В спецификации TLS 1.3 это значение называется ContentType.
• Второй и третий байты имеют значения 03 и 01 соответственно. По историческим причинам эти байты фиксированы и не являются уникальными для TLS версии 1.3. В спецификации это двухбайтовое значение называется ProtocolVersion.
• Четвертый и пятый байты кодируют длину передаваемых данных как 16-битное целое число, которое не может превышать `2^14` байтов (16 Кбайт).
• Остальные байты являются передаваемыми данными (полезной нагрузкой), длина которых равна значению, закодированному четвертым и пятым байтами записи.

> **Примечание.** Структура записи TLS сравнительно проста. Как вы увидели, заголовок записи TLS содержит всего три поля. Для сравнения, пакет IPv4 содержит перед полезной нагрузкой 14 полей, а сегмент TCP - 13.

Когда первый байт записи TLS 1.3 (ContentType) имеет значение 23, аутентифицированный шифр шифрует и аутентифицирует ее полезную нагрузку. Полезная нагрузка состоит из шифртекста, за которым следует тег аутентификации; принимающая сторона соответственно расшифровывает их и проверяет. Получатель знает, каким шифром и ключом расшифровывать, благодаря волшебству TLS: если получена зашифрованная запись TLS, шифр и ключ уже известны, поскольку они устанавливаются при выполнении протокола рукопожатия.

Одноразовые значения

В отличие от многих других протоколов, например инкапсуляции полезной нагрузки безопасности (Encapsulating Security Payload, ESP) в IPsec, записи TLS не указывают одноразовое значение, которое будет использовать аутентифицированный шифр.

Одноразовые значения для шифрования и расшифрования записей TLS формируются из 64-битных порядковых номеров, которые каждая сторона хранит локально и увеличивает для каждой новой записи. Когда клиент шифрует данные, он формирует одноразовое значение, выполняя XOR порядкового номера со значением `client_write_iv`, которое само сформировано из общего секрета. Сервер выбирает одноразовые значения при передаче данных похожим образом, но со значением `server_write_iv`.

Например, если вы передаете три записи TLS, то сформируете одноразовое значение из 0 для первой записи, из 1 для второй и из 2 для третьей; если затем получите три записи, то также используете одноразовые значения 0, 1 и 2 в этом порядке. Повторное использование одних порядковых номеров для шифрования отправляемых данных и расшифрования получаемых данных не является слабостью, поскольку они объединяются операцией XOR с разными константами (`client_write_iv` и `server_write_iv`) и для каждого направления используются разные секретные ключи.

Дополнение нулями

Записи TLS 1.3 поддерживают дополнение нулями, которое ослабляет атаки путем анализа трафика. Атакующие используют анализ трафика для извлечения сведений из его закономерностей - времени, объема переданных данных и так далее. Например, поскольку шифртексты имеют примерно тот же размер, что и открытые тексты, атакующие даже при стойком шифровании могут определить приблизительный размер сообщений, просто посмотрев на длину шифртекста.

Дополнение нулями добавляет нули к открытому тексту, увеличивая размер шифртекста и заставляя наблюдателей считать зашифрованное сообщение длиннее, чем оно есть на самом деле.

Протокол рукопожатия TLS

Рукопожатие - основа протокола согласования TLS, процесс, в ходе которого клиент и сервер устанавливают общие секретные ключи для начала защищенного обмена данными. Во время рукопожатия TLS клиент и сервер играют разные роли. Клиент предлагает несколько конфигураций (версию TLS и набор шифров в порядке предпочтения), а сервер выбирает используемую конфигурацию. Сервер должен учитывать предпочтения клиента. Для обеспечения совместимости реализаций, чтобы любой сервер TLS 1.3 мог прочитать данные TLS 1.3, отправленные любым клиентом TLS 1.3 (даже если тот использует другую библиотеку или язык программирования), спецификации TLS 1.3 также описывают формат отправляемых данных.

На рисунке 13-1 показан обмен данными в процессе рукопожатия, описанный спецификациями TLS 1.3.

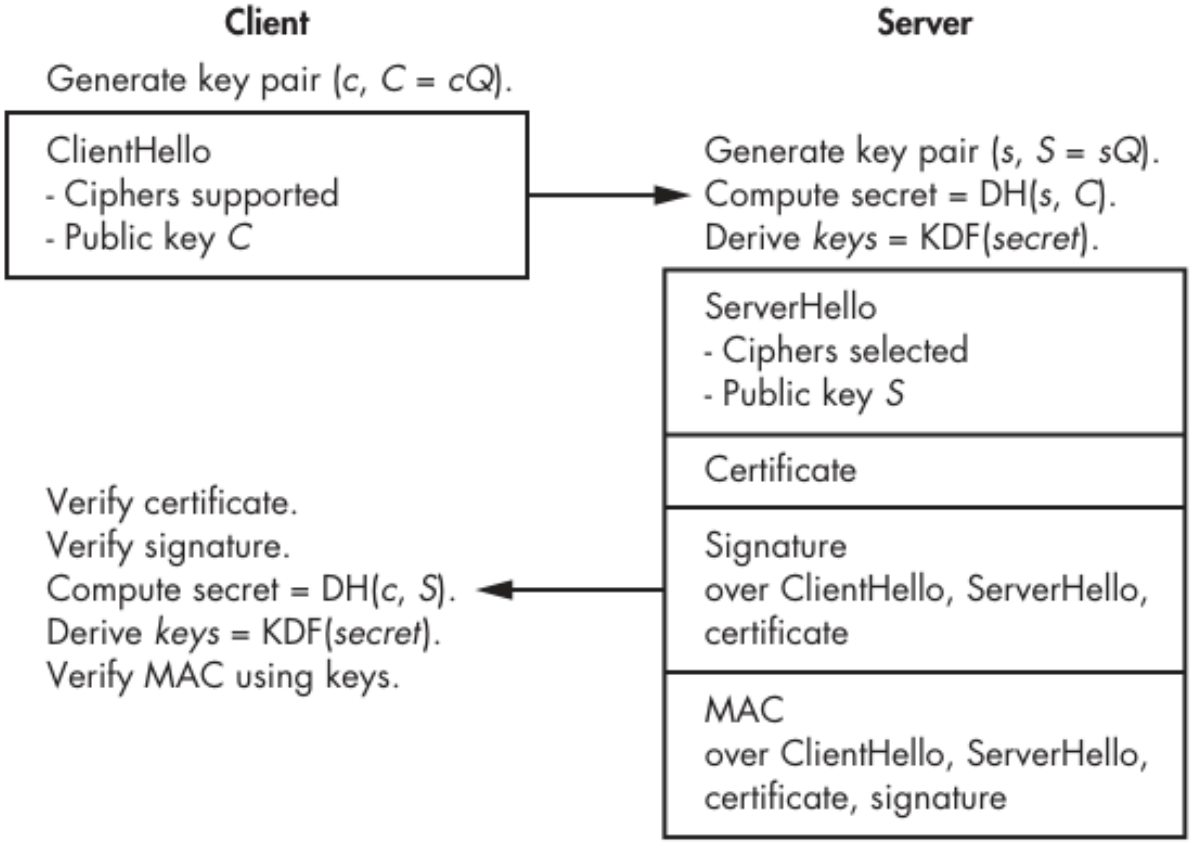

Рисунок 13-1. Процесс рукопожатия TLS 1.3

При рукопожатии TLS 1.3 клиент отправляет серверу сообщение следующего смысла: «Я хочу установить с вами TLS-соединение. Вот шифры, которые я поддерживаю для шифрования записей TLS, и вот открытый ключ Диффи-Хеллмана». Открытый ключ должен быть сгенерирован специально для этого сеанса TLS, а связанный с ним закрытый ключ клиент сохраняет. Сообщение клиента также включает случайное значение длиной 32 байта и необязательные сведения (например, дополнительные параметры). Первое сообщение, ClientHello, при передаче в виде последовательности байтов должно соответствовать особому формату, заданному спецификацией TLS 1.3.

Сервер получает ClientHello, проверяет правильность его формата и отвечает сообщением ServerHello, насыщенным сведениями. Обычно при подключении к сайту HTTPS оно содержит шифр для шифрования записей TLS, открытый ключ Диффи-Хеллмана, 32-байтовое случайное значение (рассматриваемое в разделе «Защита от понижения версии»), сертификат, подпись всей предшествующей информации в сообщениях ClientHello и ServerHello (вычисленную закрытым ключом, связанным с открытым ключом сертификата), а также MAC той же информации и подписи. MAC вычисляется симметричным ключом, сформированным из общего секрета Диффи-Хеллмана, который сервер вычисляет из своего закрытого ключа Диффи-Хеллмана и открытого ключа клиента.

Получив ServerHello, клиент проверяет действительность сертификата, подпись, вычисляет общий секрет Диффи-Хеллмана и формирует из него симметричные ключи, а также проверяет MAC сервера. Когда все проверено, клиент готов отправлять серверу зашифрованные сообщения.

> **Примечание.** TLS 1.3 поддерживает множество параметров и расширений, поэтому его поведение может отличаться. Например, рукопожатие TLS 1.3 можно настроить так, чтобы оно требовало сертификат клиента, позволяя серверу проверить личность клиента. TLS 1.3 также поддерживает рукопожатие с предварительно распределенными ключами.

Посмотрим, как это работает на практике. Допустим, вы развернули TLS 1.3 для защищенного доступа к сайту nostarch.com. Когда браузер (клиент) открывает этот сайт, он отправляет его серверу сообщение ClientHello, включающее поддерживаемые шифры. Сайт отвечает сообщением ServerHello и сертификатом, содержащим открытый ключ, связанный с доменом `www.nostarch.com`. Клиент проверяет действительность сертификата с помощью одного из центров сертификации, встроенных в браузер (полученный сертификат должен быть подписан доверенным центром сертификации, сертификат которого для успешной проверки должен находиться в хранилище сертификатов браузера или операционной системы). После успешного прохождения всех проверок браузер запрашивает у сервера `www.nostarch.com` начальную страницу сайта.

После успешного рукопожатия TLS 1.3 весь обмен данными между клиентом и сервером зашифрован и аутентифицирован. Подслушивающий может узнать, что клиент с некоторым IP-адресом общается с сервером по другому IP-адресу, и наблюдать за передаваемым зашифрованным содержимым, но не может узнать лежащий в его основе открытый текст или изменить зашифрованные сообщения (если он это сделает, принимающая сторона заметит вмешательство, поскольку сообщения аутентифицируются). Для многих приложений такой безопасности достаточно.

Криптографические алгоритмы TLS 1.3

TLS 1.3 использует алгоритмы аутентифицированного шифрования, функцию формирования ключа (хеш-функцию, формирующую секретные ключи из общего секрета), а также операцию Диффи-Хеллмана. Но как именно они работают, какие алгоритмы применяются и насколько они безопасны?

Что касается выбора аутентифицированных шифров, TLS 1.3 поддерживает только три алгоритма: AES-GCM, AES-CCM (режим, немного менее эффективный, чем GCM) и потоковый шифр ChaCha20 в сочетании с MAC Poly1305 (как определено в RFC 7539). Поскольку TLS 1.3 не позволяет использовать небезопасную длину ключа, например 64 или 80 битов, секретный ключ может иметь длину либо 128 битов (AES-GCM или AES-CCM), либо 256 битов (AES-GCM или ChaCha20-Poly1305).

Операция формирования ключа (KDF) на рисунке 13-1 основана на HKDF - конструкции на основе HMAC (см. главу 7), определенной в RFC 5869 и использующей хеш-функцию SHA-256 или SHA-384.

Варианты выполнения операции Диффи-Хеллмана (основы рукопожатия TLS 1.3) ограничены криптографией на эллиптических кривых и мультипликативной группой целых чисел по модулю простого числа (как в традиционном Диффи-Хеллмане). Но использовать любую эллиптическую кривую или группу нельзя: среди поддерживаемых кривых три кривые NIST, а также Curve25519 (см. главу 12) и Curve448, обе определенные в RFC 7748. TLS 1.3 также поддерживает DH над группами целых чисел, в отличие от эллиптических кривых. Поддерживаются группы пяти размеров, определенных в RFC 7919: 2048, 3072, 4096, 6144 и 8192 бита.

В теории 2048-битная группа может быть самым слабым звеном TLS 1.3. Тогда как остальные варианты обеспечивают не менее 128 битов безопасности, считается, что 2048-битный Диффи-Хеллман обеспечивает менее 100 битов безопасности. Поэтому поддержку 2048-битной группы можно считать несогласованной с другими проектными решениями TLS 1.3. На практике 100-битную безопасность взломать примерно так же трудно, как 128-битную, то есть практически невозможно.

Улучшения TLS 1.3 по сравнению с TLS 1.2

TLS 1.3 сильно отличается от своего предшественника. Во-первых, из него удалены слабые алгоритмы, такие как MD5, SHA-1, RC4 и AES в режиме CBC. Кроме того, если TLS 1.2 зачастую защищал записи сочетанием шифра и MAC (например, HMAC-SHA-1) в конструкции «сначала MAC, затем шифрование», то TLS 1.3 поддерживает только более эффективные и безопасные аутентифицированные шифры. TLS 1.3 также отказался от согласования кодировки точек эллиптических кривых и определяет один формат точки для каждой кривой.

В TLS 1.3 удалены возможности версии 1.2, ослаблявшие протокол, а общая сложность протокола и тем самым поверхность атаки уменьшены. Например, TLS 1.3 отказался от необязательного сжатия данных - возможности, которая позволяла проводить атаку CRIME на TLS 1.2. Эта атака использовала тот факт, что длина сжатой версии сообщения раскрывает сведения о его содержимом.

Но TLS 1.3 также принес новые возможности, делающие соединения безопаснее или эффективнее. Я рассмотрю три из них: защиту от понижения версии, рукопожатие за один круговой обмен и возобновление сеанса.

Защита от понижения версии

Защита TLS 1.3 от понижения версии противодействует атакам с понижением версии, при которых атакующий заставляет клиента и сервер использовать более слабую, чем 1.3, версию TLS. Для проведения такой атаки атакующий вынуждает сервер применить более слабую версию TLS, перехватывая и изменяя сообщение ClientHello так, чтобы сообщить серверу, будто клиент не поддерживает TLS 1.3. После этого атакующий может воспользоваться уязвимостями ранних версий TLS.

Для противодействия атакам с понижением версии сервер TLS 1.3 использует три вида шаблонов в 32-байтовом случайном значении, отправляемом в сообщении ServerHello, чтобы обозначить запрошенный тип соединения. Шаблон должен соответствовать запросу клиента на определенный вид соединения TLS. Если клиент получает неправильный шаблон, он понимает, что происходит неладное.

В частности, если клиент запрашивает соединение TLS 1.2, первые 8 из 32 байтов получают значения `44 4F 57 4E 47 52 44 01`, а при запросе соединения TLS 1.1 - `44 4F 57 4E 47 52 44 00`. Однако при запросе соединения TLS 1.3 эти первые 8 байтов должны быть случайными. Например, если клиент отправляет ClientHello с запросом соединения TLS 1.3, но сетевой атакующий изменяет его на запрос соединения TLS 1.1, клиент, получив ServerHello с неправильным шаблоном, понимает, что его сообщение ClientHello было изменено. (Атакующий не может произвольно изменить 32-байтовое случайное значение сервера, поскольку оно подписано криптографически.)

Рукопожатие за один круговой обмен

При обычном рукопожатии TLS 1.2 клиент отправляет серверу некоторые данные, ожидает ответа, затем отправляет дополнительные данные и снова ожидает ответа сервера, прежде чем начать отправлять зашифрованные сообщения. Задержка составляет два времени кругового обмена (round-trip time, RTT). В отличие от него, рукопожатие TLS 1.3 занимает одно время кругового обмена (см. рисунок 13-1). Экономия времени может составлять сотни миллисекунд. Это существенно, если учесть, что серверы популярных служб обрабатывают тысячи соединений в секунду.

Возобновление сеанса

TLS 1.3 быстрее TLS 1.2, но его можно ускорить еще сильнее (на величину порядка сотен миллисекунд), полностью исключив круговые обмены, предшествующие зашифрованному сеансу. Прием состоит в использовании возобновления сеанса, которое задействует предварительно распределенный ключ, переданный между клиентом и сервером в предыдущем сеансе, чтобы запустить новый. Возобновление сеанса дает два главных преимущества: клиент может начать шифрование немедленно, а в последующих сеансах не требуется использовать сертификаты.

Работа возобновления сеанса показана на рисунке 13-2.

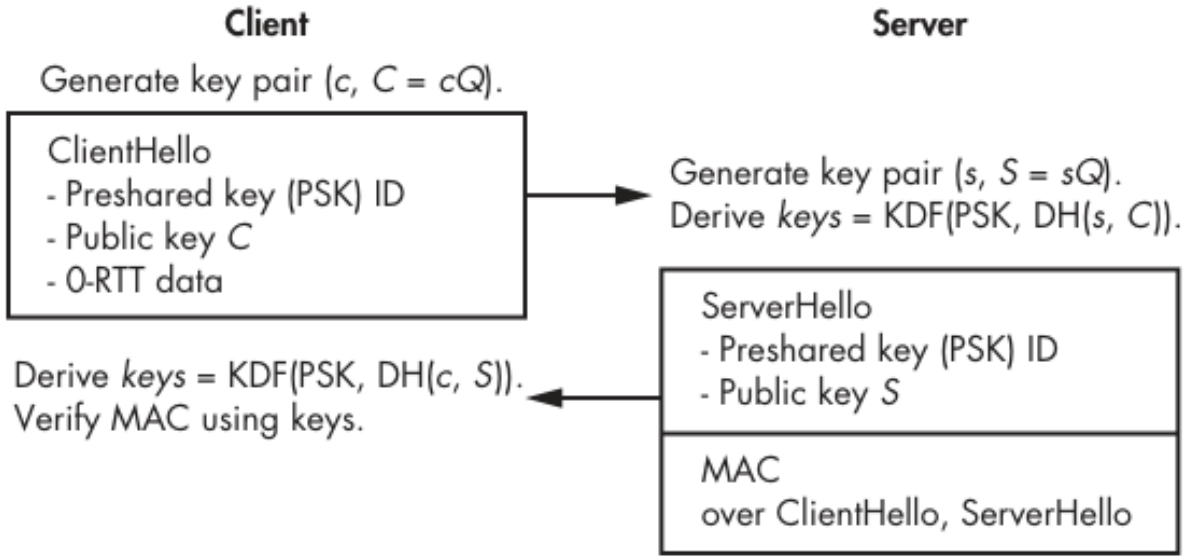

Рисунок 13-2. Рукопожатие при возобновлении сеанса TLS 1.3, где данные 0-RTT - это данные возобновления сеанса, отправленные вместе с ClientHello

Сначала клиент отправляет сообщение ClientHello, включающее идентификатор ключа, уже общего с сервером (называемого PSK, от preshared key, предварительно распределенного ключа), вместе со свежим открытым ключом DH. Клиент также может включить в первое сообщение зашифрованные данные (называемые данными 0-RTT). Отвечая на ClientHello, сервер предоставляет MAC всего обмена данными. Клиент проверяет MAC и понимает, что общается с тем же сервером, что и раньше, поэтому проверка сертификата становится в некоторой мере излишней. Клиент и сервер выполняют согласование ключей Диффи-Хеллмана, как при обычном рукопожатии, а последующие сообщения шифруются ключами, зависящими как от PSK, так и от только что вычисленного общего секрета Диффи-Хеллмана.

Сильные стороны безопасности TLS

Оценим сильные стороны TLS 1.3 относительно двух основных понятий безопасности из главы 11: аутентификации и прямой секретности.

Аутентификация

Во время рукопожатия TLS 1.3 сервер аутентифицируется перед клиентом с помощью механизма сертификатов. Однако клиент не аутентифицируется; клиент может пройти аутентификацию в серверном приложении (например, Gmail), предоставив имя пользователя и пароль в записи TLS после выполнения рукопожатия. Если клиент уже установил сеанс с удаленной службой, он может аутентифицироваться, отправив защищенный файл cookie, который можно передать только через TLS-соединение.

В некоторых случаях клиенты могут аутентифицироваться перед сервером с помощью механизма на основе сертификатов, похожего на механизм аутентификации сервера перед клиентом: клиент отправляет серверу сертификат клиента, который тот проверяет, прежде чем авторизовать клиента. Однако сертификаты клиентов используются редко, поскольку усложняют работу и клиентам, и серверу (то есть издателю сертификатов): клиентам приходится выполнять сложные операции, чтобы встроить сертификат в свою систему и защитить его закрытый ключ, а издатель, помимо прочих требований, должен гарантировать, что сертификат получили только авторизованные клиенты.

Прямая секретность

Вспомним из раздела «Протоколы согласования ключей» главы 11, что согласование ключей обеспечивает прямую секретность, если при компрометации настоящего сеанса не компрометируются прежние. В модели утечки данных компрометируются только временные секреты, тогда как в модели взлома раскрываются долгосрочные секреты.

К счастью, прямая секретность TLS 1.3 сохраняется и при утечке данных, и при взломе. В модели утечки данных атакующий восстанавливает временные секреты, например сеансовые ключи или закрытые ключи Диффи-Хеллмана конкретного сеанса (значения `c`, `s`, `secret` и `keys` на рисунке 13-1). Однако с их помощью он может расшифровать обмен данными только в настоящем сеансе, но не в предыдущих, поскольку там использовались другие значения `c` и `s` (и получались другие ключи).

В модели взлома атакующий также восстанавливает долгосрочные секреты (а именно закрытый ключ, соответствующий открытому ключу сертификата). Однако для расшифрования прежних сеансов он не полезнее временных секретов, поскольку этот закрытый ключ служит лишь для аутентификации сервера, и прямая секретность снова сохраняется.

На практике, если атакующий компрометирует клиентскую машину и получает доступ ко всей ее памяти, он может восстановить из памяти сеансовые ключи TLS и секреты текущего сеанса. Но что еще важнее, если прежние ключи все еще находятся в памяти, атакующий может найти их и расшифровать предыдущие сеансы, тем самым обойдя теоретическую прямую секретность. Поэтому для обеспечения прямой секретности реализация TLS должна надлежащим образом стирать ключи из памяти, когда они больше не используются, обычно записывая в память нули.

Что может пойти не так

TLS 1.3 соответствует требованиям к универсальному протоколу защищенного обмена данными, но не является неуязвимым. Как и любая система безопасности, он может дать сбой в определенных обстоятельствах (например, если предположения разработчиков о реальных атаках неверны). К сожалению, даже последняя версия TLS 1.3, настроенная с самыми безопасными шифрами, может быть скомпрометирована. Например, безопасность TLS 1.3 основана на предположении, что все три стороны (клиент, сервер и центр сертификации) ведут себя честно, но что произойдет, если одна сторона скомпрометирована или сама реализация TLS выполнена плохо?

Скомпрометированный центр сертификации

Корневые центры сертификации (root CA) - это организации, которым браузеры доверяют проверку сертификатов, предоставляемых удаленными узлами. Например, если браузер принимает сертификат от `www.google.com`, предполагается, что доверенный CA проверил законность владельца сертификата. Браузер проверяет сертификат по выпущенной CA подписи. Поскольку закрытый ключ, необходимый для создания этой подписи, известен только CA, считается, что другие не могут создавать действительные сертификаты от имени CA. Очень часто сертификат сайта подписан не корневым CA, а промежуточным CA, связанным с корневым CA цепочкой сертификатов.

Если закрытый ключ CA скомпрометирован, атакующий может с помощью этого ключа создать сертификат для любого URL, например в домене `google.com`, без разрешения Google. Затем атакующий может использовать такие сертификаты, чтобы выдать себя за законный сервер или поддомен вроде `mail.google.com` и перехватывать учетные данные и обмен данными пользователя. Именно это произошло в 2011 году, когда атакующий взломал сеть нидерландского центра сертификации DigiNotar и создал выглядевшие законными сертификаты. Атакующий использовал эти поддельные сертификаты для нескольких служб Google.

Скомпрометированный сервер

Если сервер скомпрометирован и полностью контролируется атакующим, потеряно все: сервер хранит сеансовые ключи, являясь конечной точкой TLS-соединения. Атакующий видит все отправляемые данные до шифрования и все получаемые после расшифрования. Вероятно, он также получит закрытый ключ сервера, что позволит ему выдавать себя за законный сервер с помощью собственного вредоносного сервера. В этом случае TLS вас не спасет.

К счастью, такие катастрофы безопасности редко наблюдаются в известных приложениях вроде Gmail и iCloud, которые хорошо защищены и иногда хранят закрытые ключи в отдельном модуле безопасности, например аппаратном модуле безопасности (hardware security module, HSM), непосредственно или через приложение системы управления ключами (key management system, KMS).

Атаки на веб-приложения посредством таких уязвимостей, как внедрение запросов к базе данных и межсайтовый скриптинг, распространены больше, поскольку почти не зависят от безопасности TLS и проводятся атакующими через законное TLS-соединение. Такие атаки могут привести к компрометации имен пользователей, паролей и других данных.

Скомпрометированный клиент

Безопасность TLS также оказывается под угрозой, когда клиент, например браузер, скомпрометирован удаленным атакующим. Скомпрометировав клиент, атакующий способен перехватывать сеансовые ключи, читать любые расшифрованные данные и так далее. Он даже может установить в системе клиента мошеннический сертификат CA, чтобы та незаметно принимала сертификаты, которые в обычных условиях недействительны, позволяя атакующему перехватывать TLS-соединения.

Различие между сценариями со скомпрометированным CA или сервером и сценарием со скомпрометированным клиентом состоит в том, что в последнем случае страдает только конкретный клиент, а не потенциально все клиенты.

Ошибки в реализациях

Как и любой криптографический компонент, TLS может дать сбой из-за ошибок в реализации. Самый яркий пример ошибки TLS - Heartbleed (см. рисунок 13-3), переполнение буфера в реализации OpenSSL второстепенной возможности TLS, называемой heartbeat. Heartbleed была обнаружена в 2014 году независимо исследователем Google и компанией Codenomicon и затронула миллионы серверов и клиентов TLS.

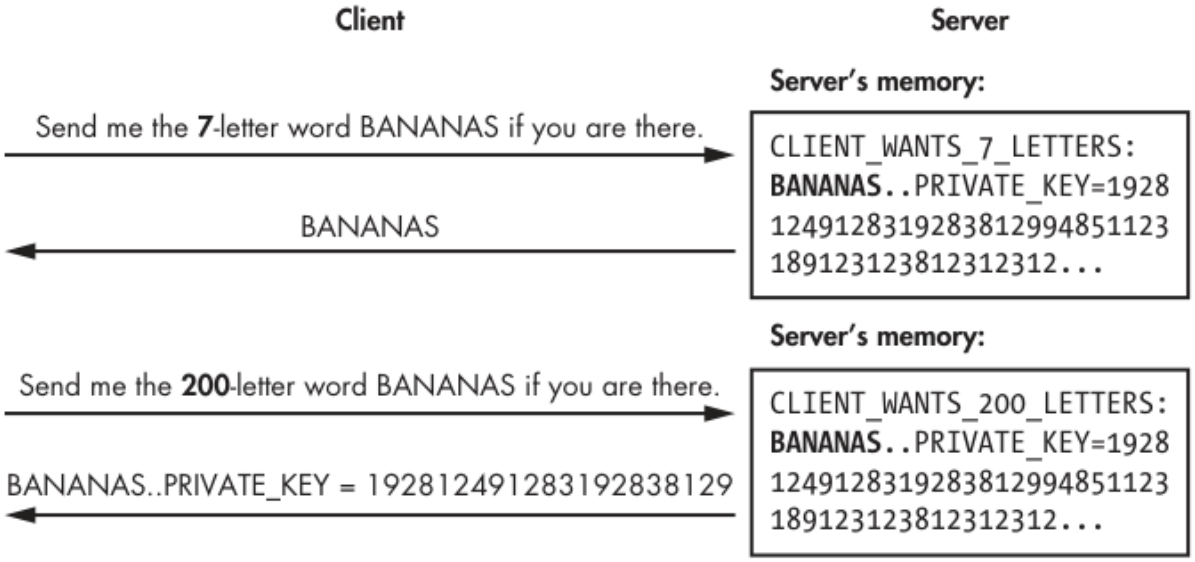

Рисунок 13-3. Ошибка Heartbleed в реализациях TLS на основе OpenSSL

Сначала клиент отправляет серверу буфер вместе с его длиной, чтобы проверить, находится ли сервер в сети. В этом примере буфер - строка `BANANAS`, и клиент явно сообщает, что длина слова составляет семь букв. Сервер читает семибуквенное слово и возвращает его клиенту.

Проблема в том, что сервер не проверяет правильность длины и пытается прочитать столько символов, сколько указал клиент. Поэтому, если клиент передает длину, превышающую фактическую длину строки, сервер считывает из памяти слишком много данных и возвращает их клиенту вместе с лишними данными, которые могут содержать конфиденциальные сведения, например закрытые ключи или сеансовые файлы cookie.

Ошибка Heartbleed стала потрясением. Чтобы избежать подобных ошибок в будущем, OpenSSL и другие крупные реализации TLS теперь проходят строгую проверку кода и используют автоматизированные инструменты, например фаззеры, для выявления потенциальных проблем.

Дополнительная литература

Эта глава не является исчерпывающим руководством по TLS; возможно, вам захочется глубже изучить историю TLS, его прежние уязвимости и последнюю версию. Полная спецификация TLS 1.3, размещенная на домашней странице рабочей группы TLS (TLSWG) по адресу tlswg.org, содержит все сведения о протоколе (хотя необязательно объясняет лежащие в его основе причины).

Я также рекомендую познакомиться с основными протоколами, использующими TLS, например QUIC (применяется в соединениях между Chrome и серверами Google) и SRTP (используется для видеоконференций и потокового трафика).

Кроме того, существуют две важные инициативы, связанные с развертыванием TLS:

• Тест TLS от SSL Labs (ssllabs.com) - бесплатная служба Qualys, позволяющая проверить конфигурацию TLS браузера или сервера и получить оценку безопасности, а также предложения по улучшению. Если вы настраиваете собственный сервер TLS, воспользуйтесь этим тестом, чтобы убедиться в безопасности.
• Let's Encrypt (letsencrypt.org) - некоммерческая организация, предлагающая службу «автоматического волшебного» развертывания TLS на HTTP-серверах. Она включает средства автоматической генерации сертификата и настройки сервера TLS и поддерживает все распространенные веб-серверы и операционные системы.

Глава 14. Квантовая и постквантовая криптография

В этой главе мы рассмотрим будущее криптографии на горизонте, скажем, столетия или более - будущее, в котором могут существовать квантовые компьютеры. Квантовые компьютеры используют явления квантовой физики, чтобы выполнять алгоритмы, отличающиеся от привычных нам. Хотя больших квантовых компьютеров пока нет, потенциально они способны взломать RSA, Диффи-Хеллман и криптографию на эллиптических кривых - всю криптографию с открытым ключом, развернутую или стандартизированную на момент написания книги.

Для защиты от риска квантовых вычислений исследователи криптографии разработали альтернативные постквантовые алгоритмы с открытым ключом. В 2015 году АНБ призвало перейти на квантовоустойчивые алгоритмы, а в 2017 году NIST начал процесс стандартизации постквантовых алгоритмов, которые были объявлены частью новых стандартов в 2022 году.

Эта глава дает нетехнический обзор принципов квантовых компьютеров и позволяет мельком взглянуть на постквантовые алгоритмы. Здесь есть немного математики, но лишь базовая арифметика и линейная алгебра, поэтому не тревожьтесь из-за непривычных обозначений.

Как работают квантовые компьютеры

Квантовые вычисления используют квантовую физику, чтобы вычислять иначе и выполнять задачи, недоступные классическим компьютерам, например эффективно взламывать RSA и криптографию на эллиптических кривых. Квантовый компьютер - не сверхбыстрый обычный компьютер. На самом деле квантовые компьютеры не могут эффективно решить любую задачу, слишком трудную для классического компьютера, например полный перебор или NP-трудные задачи.

Квантовая механика - раздел физики, изучающий поведение субатомных частиц, которые ведут себя истинно случайным образом, - служит основой квантовых компьютеров. В отличие от классических компьютеров, работающих с битами, равными либо 0, либо 1, квантовые компьютеры работают с квантовыми битами (кубитами), которые могут быть «одновременно и 0, и 1». Это неоднозначное состояние называется суперпозицией; кавычки указывают на упрощение. В этом микроскопическом мире физики обнаружили, что такие частицы, как электроны и фотоны, ведут себя крайне нелогично: прежде чем вы наблюдаете электрон, он находится не в определенном месте пространства, а сразу в нескольких местах (то есть в состоянии суперпозиции). После наблюдения - операции, называемой измерением, - он останавливается в фиксированном случайном месте и больше не находится в суперпозиции; квантовое состояние коллапсирует. Это позволяет создавать кубиты квантового компьютера наряду с явлениями запутанности и интерференции.

Квантовые компьютеры работают благодаря запутанности, при которой две частицы связаны (запутаны) так, что наблюдение значения одной дает значение другой, даже если частицы разделены огромным расстоянием (километрами или даже световыми годами). Парадокс Эйнштейна-Подольского-Розена (ЭПР) иллюстрирует это поведение, из-за которого Альберт Эйнштейн поначалу отвергал квантовую механику. (Подробное объяснение см. на plato.stanford.edu.)

Интерференция также имеет решающее значение для работы квантовых компьютеров. Благодаря этому свойству частицы из-за своей волновой природы могут объединять или взаимно уничтожать воздействие друг друга, что наглядно показывает знаменитый опыт с двумя щелями. Квантовые вычисления используют интерференцию так, чтобы «волны» правильных решений усиливали друг друга, а неправильные решения взаимно уничтожались.

Объясняя работу квантового компьютера, я буду различать собственно квантовый компьютер (аппаратное обеспечение, включая его квантовые биты) и квантовые алгоритмы (работающее на нем программное обеспечение, состоящее из квантовых вентилей).

Квантовые биты

Кубиты или их группы можно характеризовать амплитудами - числами, похожими на вероятности, но не являющимися вероятностями в точности. Тогда как вероятность - число от 0 до 1, амплитуда является комплексным числом вида `a+bi` (то есть `a+b* i`), где `a` и `b` - вещественные числа, а `i` - мнимая единица. Число `i` используется для образования мнимых чисел вида `bi`, где `b` - вещественное число. При умножении `i` на вещественное число получается другое мнимое число, а при умножении на себя - значение -1 (то есть `i^2=-1`).

В отличие от вещественных чисел, которые можно представить принадлежащими прямой (см. рисунок 14-1), комплексные числа принадлежат плоскости (двумерному пространству), как показано на рисунке 14-2.

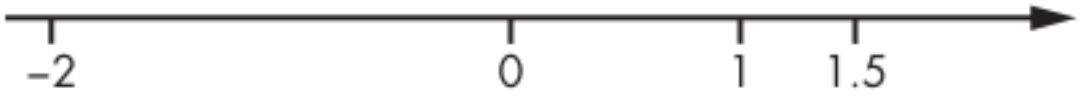

Рисунок 14-1. Вещественные числа как точки на бесконечной прямой

На рисунке 14-2 ось `x` соответствует `a` в `a+bi`, ось `y` - значению `b`, а пунктирные линии - вещественной и мнимой частям каждого числа. Например, вертикальная пунктирная линия, идущая от точки `3+2i` вниз к 3, имеет длину две единицы (значение 2 в мнимой части `2i`).

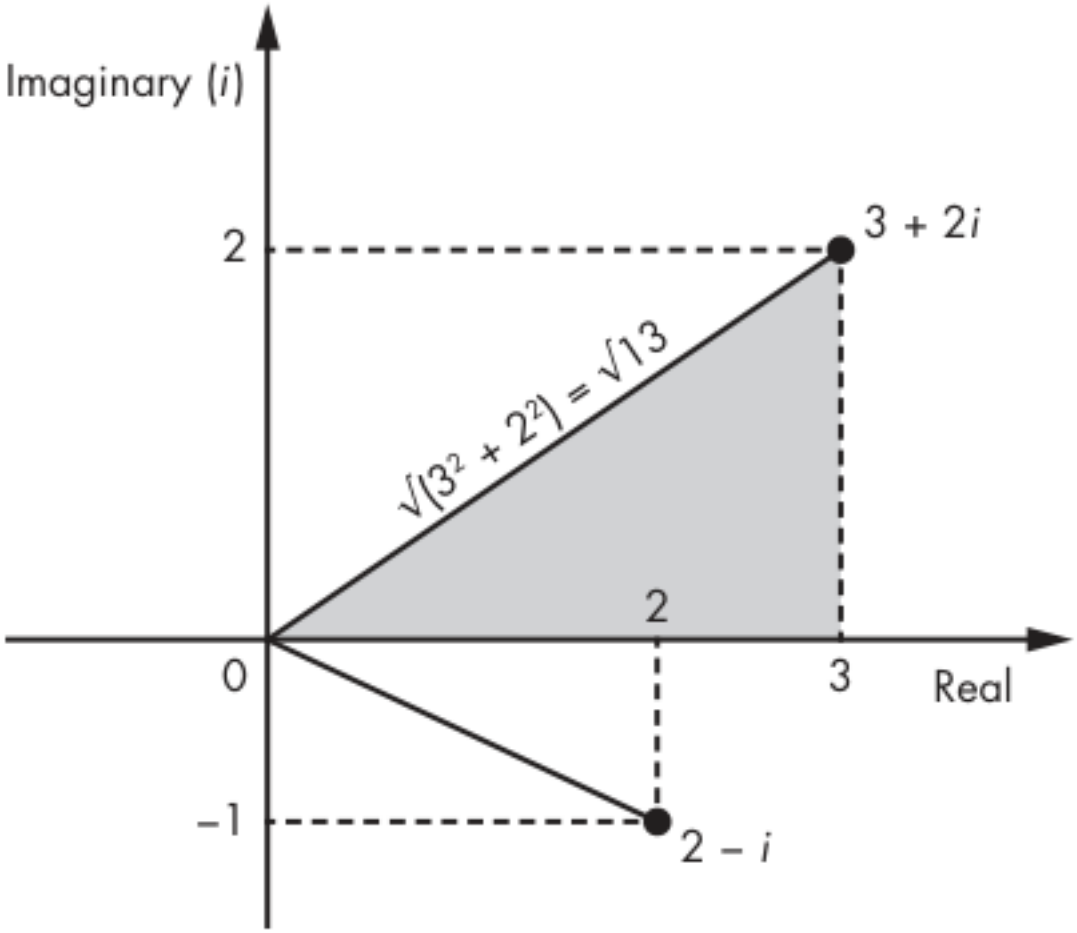

Рисунок 14-2. Комплексные числа как точки в двумерном пространстве

С помощью теоремы Пифагора можно вычислить длину отрезка, идущего от начала координат (0) к точке `a+bi`, рассматривая его как гипотенузу треугольника. Длина гипотенузы равна квадратному корню из суммы квадратов координат точки, то есть `sqrt(a^2+b^2)`, и называется модулем комплексного числа `a+bi`. Модуль обозначается `|a+bi|` и может использоваться как длина комплексного числа.

В квантовом компьютере регистры состоят из одного или нескольких кубитов в состоянии суперпозиции, которое можно характеризовать набором таких комплексных чисел, или амплитуд. Но, как вы увидите, амплитудами могут быть не любые числа.

Амплитуды одного кубита

Один кубит можно характеризовать двумя амплитудами, которые обозначим `alpha` (альфа) и `beta` (бета). Тогда состояние кубита можно выразить как `alpha|0>+beta|1>`, где обозначение `| >` представляет векторы в квантовом состоянии. Эта запись означает: при наблюдении кубит окажется равным 0 с вероятностью `|alpha|^2` и 1 с вероятностью `|beta|^2`. Чтобы это были настоящие вероятности, `|alpha|^2` и `|beta|^2` должны быть числами от 0 до 1, а сумма `|alpha|^2+|beta|^2` должна равняться 1.

Например, пусть имеется кубит `Psi` (пси) с амплитудами `alpha=1/sqrt(2)` и `beta=1/sqrt(2)`. Он выражается следующим образом:

```
Psi=(1/sqrt(2))|0>+(1/sqrt(2))|1>=(|0>+|1>)/sqrt(2).
```

В кубите Psi значение 0 имеет амплитуду 1/sqrt(2), и значение 1 имеет ту же амплитуду 1/sqrt(2). Чтобы получить фактическую вероятность из амплитуды, вычислим модуль 1/sqrt(2) (он равен 1/sqrt(2), поскольку мнимой части нет), а затем возведем его в квадрат: (1/sqrt(2))^2=1/2. Это означает, что при наблюдении кубита Psi вероятность увидеть 0 равна 1/2 и вероятность увидеть 1 такая же.

Теперь рассмотрим кубит Phi (фи), где

```
Phi=(i/sqrt(2))|0>-(1/sqrt(2))|1>=(i|0>-|1>)/sqrt(2),
```

или |Phi>=(i/sqrt(2),-1/sqrt(2)).

Кубит Phi принципиально отличается от Psi: в отличие от Psi, где амплитуды имеют равные значения, у кубита Phi разные амплитуды alpha=i/sqrt(2) (положительное мнимое число) и beta=-1/sqrt(2) (отрицательное вещественное число). Однако при наблюдении Phi вероятность увидеть 0 или 1 равна 1/2, как и для Psi. Вычислим вероятность увидеть 0 по приведенным выше правилам:

```
|alpha|^2=(sqrt((1/sqrt(2))^2))^2=1/(sqrt(2))^2=1/2.
```

Заметьте: поскольку alpha=i/sqrt(2), его можно записать как a+bi при a=0 и b=1/sqrt(2), а вычисление |alpha|=sqrt(a^2+b^2) дает 1/sqrt(2).

Разные кубиты могут одинаково вести себя для наблюдателя (иметь одинаковую вероятность увидеть 0), но обладать разными амплитудами. Это означает, что фактические вероятности увидеть 0 или 1 характеризуют кубит лишь частично; это похоже на наблюдение тени предмета на стене, которое дает представление о его ширине и высоте, но не о глубине. В случае кубитов этим скрытым измерением является значение амплитуды: положительное оно или отрицательное? Вещественное или мнимое число?

> **Примечание.** Для упрощения обозначений кубит часто записывают парой его амплитуд (alpha,beta). Поэтому предыдущий пример можно записать как |Psi>=(1/sqrt(2),1/sqrt(2)).

Амплитуды групп кубитов

Как понимать несколько кубитов? Например, восемь кубитов могут образовать квантовый байт, если квантовые состояния этих восьми кубитов связаны запутанностью. Такой квантовый байт можно описать следующим образом, где alpha - амплитуды, связанные с каждым из 256 возможных значений группы из восьми кубитов:

```
alpha_0|00000000>+
alpha_1|00000001>+
alpha_2|00000010>+
alpha_3|00000011>+
...+
alpha_255|11111111>.
```

Заметьте, что должно выполняться |alpha_0|^2+|alpha_1|^2+...+|alpha_255|^2=1, чтобы сумма всех вероятностей равнялась 1.

Эту группу из восьми кубитов можно рассматривать как набор из 2^8=256 амплитуд, поскольку у нее 256 возможных конфигураций, каждая со своей амплитудой. Однако в физической реальности существует восемь физических объектов, а не 256. Эти 256 амплитуд - неявная характеристика группы из восьми кубитов; каждое из этих 256 чисел может принимать любое из бесконечного множества значений. В общем случае группу из n кубитов можно характеризовать набором из 2^n комплексных чисел, число которых экспоненциально растет вместе с количеством кубитов.

Чтобы смоделировать эволюцию квантового состояния на классическом компьютере, необходимо хранить это экспоненциальное число амплитуд и выполнять вычисления для их изменения. Это требование - одна из основных причин, по которым классический компьютер не может эффективно моделировать квантовый: для хранения того же объема информации, который квантовая система содержит всего в `n` кубитах, требуется гигантский объем памяти (порядка `2^n`). На практике можно моделировать не более 50 или 60 кубитов в зависимости от вида вычисления.

Квантовые вентили

Понятия амплитуды и квантовых вентилей уникальны для квантовых вычислений. Квантовый вентиль по существу является преобразованием одного или нескольких кубитов и представляет собой аналог электронных вентилей в области квантовых вычислений. Тогда как классический компьютер использует регистры, память и микропроцессор для выполнения последовательности инструкций над данными, квантовый компьютер обратимо преобразует группу кубитов, применяя ряд квантовых вентилей, а затем измеряет значение одного или нескольких кубитов. Квантовые компьютеры обещают большую вычислительную мощность, поскольку всего с `n` кубитами могут воздействовать на значения `2^n` амплитуд. Это свойство имеет глубокие последствия.

С математической точки зрения квантовые алгоритмы по существу являются схемой квантовых вентилей, преобразующей набор комплексных чисел (амплитуд) перед окончательным измерением, при котором наблюдается значение одного или нескольких кубитов (см. рисунок 14-3).

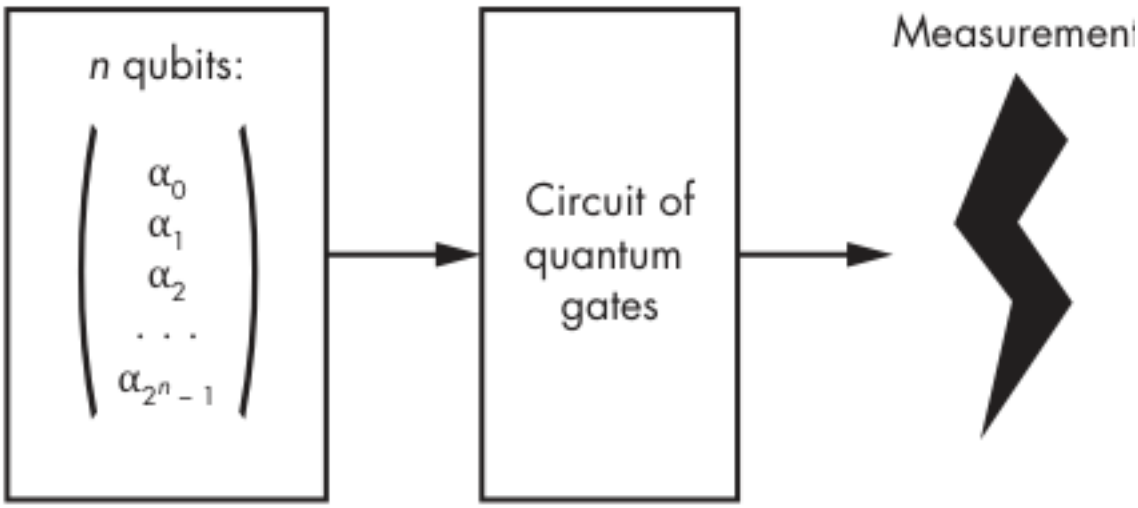

Рисунок 14-3. Принцип квантового алгоритма

Квантовые алгоритмы также называют массивами квантовых вентилей или квантовыми схемами.

Квантовые вентили как умножение матриц

В отличие от булевых вентилей классического компьютера (AND, XOR и других), квантовый вентиль воздействует на группу амплитуд так же, как матрица при умножении на вектор. Например, чтобы применить к кубиту `Phi` простейший квантовый вентиль - тождественный вентиль, - представим `I` как единичную матрицу `2*2` и умножим ее на вектор-столбец, состоящий из двух амплитуд `Phi`:

```
I|Phi>=
[1 0
0 1]
[i/sqrt(2)
-1/sqrt(2)] =
[
1* i/sqrt(2)+0*(-1/sqrt(2))
0* i/sqrt(2)+1*(-1/sqrt(2))
] =
[i/sqrt(2)
-1/sqrt(2)]
=|Phi>.
```

Результат этого умножения матрицы на вектор - другой вектор-столбец из двух элементов, где верхнее значение равно скалярному произведению первой строки матрицы `I` на входной вектор (сумме произведения первых элементов 1 и `i/sqrt(2)` и произведения вторых элементов 0 и `-1/sqrt(2)`); нижнее значение вычисляется аналогично.

Тождественный вентиль `I` почти бесполезен, потому что ничего не делает и оставляет кубит неизменным.

> **Примечание.** На практике квантовый компьютер не стал бы явно вычислять произведения матриц на векторы, поскольку матрицы были бы слишком велики. (Именно поэтому классический компьютер не может моделировать квантовые вычисления.) Вместо этого квантовый компьютер преобразовывал бы кубиты как физические частицы посредством физических преобразований, эквивалентных умножению матриц. Запутались? Вот что сказал Ричард Фейнман: «Если квантовая механика не привела вас в полное замешательство, вы ее не понимаете».

Квантовый вентиль Адамара

Один из самых полезных квантовых вентилей - вентиль Адамара, обычно обозначаемый `H`. Вентиль Адамара определяется следующим образом (обратите внимание на отрицательное значение в правой нижней позиции):

```
H=
[
1/sqrt(2) 1/sqrt(2)
1/sqrt(2) -1/sqrt(2)
].
```

Применение этого вентиля к кубиту `|Psi>=(1/sqrt(2),1/sqrt(2))` дает

```
H|Psi>=
[
1/sqrt(2) 1/sqrt(2)
1/sqrt(2) -1/sqrt(2)
]
[1/sqrt(2)
1/sqrt(2)] =
[1
0]
=|0>.
```

Применив вентиль Адамара `H` к `|Psi>`, мы получаем кубит `|0>`, в котором значение `|0>` имеет амплитуду 1, а `|1>` - амплитуду 0. Это означает, что кубит ведет себя детерминированно: при его наблюдении всегда будет виден 0 и никогда 1. Иными словами, случайность исходного кубита `|Psi>` потеряна.

Повторное применение вентиля Адамара к кубиту `|0>` дает

```
H|0>=
[
1/sqrt(2) 1/sqrt(2)
1/sqrt(2) -1/sqrt(2)
]
[1
0] =
[1/sqrt(2)
1/sqrt(2)]
=|Psi>.
```

Это возвращает нас к кубиту `|Psi>` и рандомизированному состоянию. В квантовых алгоритмах вентиль Адамара часто используется для перехода от детерминированного состояния к равномерно случайному.

Не всякая матрица является квантовым вентилем

Хотя применение квантовых вентилей можно представлять как умножение матриц, не всякая матрица соответствует квантовому вентилю. Напомним, что кубит состоит из комплексных чисел `alpha` и `beta` - амплитуд кубита, удовлетворяющих условию `|alpha|^2+|beta|^2=1`. Если после умножения кубита на матрицу получены две амплитуды, не удовлетворяющие этому условию, результат не может быть кубитом. Квантовым вентилям соответствуют только унитарные матрицы, сохраняющие свойство `|alpha|^2+|beta|^2=1`.

Унитарные матрицы (и квантовые вентили по определению) обратимы: имея результат операции, можно восстановить исходный кубит, применив обратную матрицу. Поэтому квантовые вычисления являются разновидностью обратимых вычислений.

Квантовое ускорение

Квантовое ускорение возникает, когда квантовый компьютер может решить задачу принципиально быстрее классического. Например, для поиска элемента среди `n` элементов неупорядоченного списка на классическом компьютере в среднем требуется `n/2` операций, поскольку перед нахождением искомого элемента приходится просматривать элементы списка. (В среднем элемент найдется после просмотра половины списка.) Ни один классический алгоритм не может добиться результата лучше `n/2`. Однако существует квантовый алгоритм, позволяющий найти элемент всего за `O(sqrt(n))` операций, что на порядки меньше `n/2`. Например, если `n` равно 1 000 000, то `n/2` равно 500 000, а `sqrt(n)` - 1000.

Различие между квантовыми и классическими алгоритмами количественно выражается временной сложностью, которую мы обозначаем `O()`. В предыдущем примере квантовый алгоритм работает за время `O(sqrt(n))`, тогда как классический алгоритм не может быть быстрее `O(n)`. Поскольку это различие во временной сложности обусловлено квадратной степенью, оно называется квадратичным ускорением. Хотя такое ускорение, скорее всего, существенно, существуют намного более мощные виды.

Экспоненциальное ускорение и задача Саймона

Экспоненциальные ускорения - святой Грааль квантовых вычислений. Они возникают, когда задача, требующая экспоненциального времени на классическом компьютере, например O(2^n), может быть выполнена на квантовом компьютере с полиномиальной сложностью, а именно O(n^k) для некоторого фиксированного числа k. Такое экспоненциальное ускорение способно превратить практически невозможную задачу в осуществимую. (Вспомним из главы 9, что криптографы и специалисты по теории сложности связывают экспоненциальное время с невозможным, а полиномиальное - с практичным.)

Самый яркий пример экспоненциального ускорения - задача Саймона. В этой вычислительной задаче функция f() преобразует n-битные строки в n-битные строки так, что результат f() выглядит как случайная n-битная строка, но с одним ограничением: существует секретное значение m, при котором для любых двух значений x, y равенство f(x)=f(y) выполняется тогда и только тогда, когда y=x xor m. Задача Саймона состоит в нахождении m при доступе к f() как к черному ящику.

Решение задачи Саймона классическим алгоритмом сводится к поиску коллизии, то есть значений x и y, для которых f(x)=f(y). Для этого требуется приблизительно 2^(n/2) запросов к f(). Однако на рисунке 14-4 показано, что квантовый алгоритм может решить задачу Саймона всего приблизительно за n запросов, с чрезвычайно эффективной временной сложностью O(n).

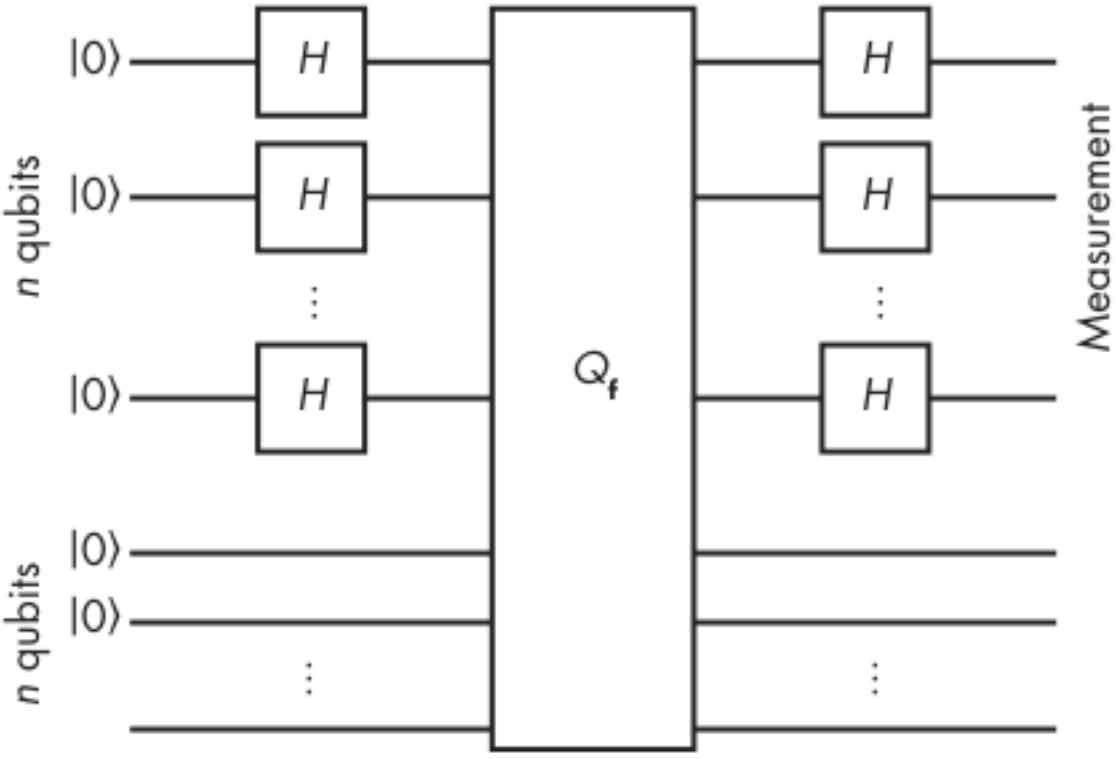

Рисунок 14-4. Квантовая схема алгоритма, эффективно решающего задачу Саймона

В квантовой схеме, решающей задачу Саймона, 2n кубитов инициализируются значением |0>, к первым n кубитам применяются вентили Адамара (H), а затем к обеим группам из n кубитов применяется вентиль Q_f. Имея две n-кубитные группы x и y, вентиль Q_f преобразует квантовое состояние |x>|y> в состояние |x>|f(x) xor y>. Иными словами, он обратимо вычисляет функцию f() над квантовым состоянием, поскольку из нового состояния можно перейти к старому, вычислив f(x) и выполнив XOR этого значения с f(x) xor y. (Объяснение того, почему это работает, выходит за рамки книги.)

Экспоненциальное ускорение для задачи Саймона можно использовать против симметричных шифров лишь в очень специфических случаях, однако в следующем разделе будут рассмотрены настоящие криптоубийственные применения квантовых вычислений.

Угроза алгоритма Шора

В 1995 году исследователь AT&T Питер Шор опубликовал поразительную статью «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer» («Полиномиальные алгоритмы факторизации простых чисел и дискретных логарифмов на квантовом компьютере»). Алгоритм Шора - квантовый алгоритм, обеспечивающий экспоненциальное ускорение при решении задач факторизации, дискретного логарифмирования (DLP) и дискретного логарифмирования на эллиптических кривых (ECDLP). Эти задачи нельзя эффективно решить классическим компьютером, но можно решить квантовым. Это означает, что квантовый компьютер способен взломать любой криптографический алгоритм, основанный на этих задачах, включая RSA, Диффи-Хеллман, криптографию на эллиптических кривых и большинство развернутых сегодня механизмов криптографии с открытым ключом (кроме перешедших на постквантовую криптографию). Иными словами, безопасность RSA или криптографии на эллиптических кривых можно было бы свести к безопасности шифра Цезаря. (Шор с тем же успехом мог назвать статью «Взлом всей криптографии с открытым ключом на квантовом компьютере».) Известный специалист по теории сложности Скотт Ааронсон назвал алгоритм Шора «одним из главных научных достижений конца XX века».

На самом деле алгоритм Шора решает более общий класс задач, чем факторизация и дискретные логарифмы. В частности, если функция `f()` периодична, то есть существует `omega` (период), при котором `f(x+omega)=f(x)` для любого `x`, алгоритм Шора эффективно найдет `omega`. (Это очень похоже на задачу Саймона, которая стала одним из главных источников вдохновения для алгоритма Шора.)

Подробное обсуждение способа, которым алгоритм Шора добивается ускорения, слишком техническое для этой книги, однако в следующем разделе показано, как с помощью алгоритма Шора решать задачи факторизации и дискретного логарифмирования (см. главу 9), трудные задачи, лежащие в основе RSA и Диффи-Хеллмана.

Задача факторизации

Допустим, требуется факторизовать большое число `N=pq`. Факторизовать `N` легко, если можно вычислить период `a^x mod N` при некоторой константе `a`. Классическому компьютеру это сделать трудно, а квантовому легко. Сначала выберите случайное число `a`, меньшее `N`, и попросите алгоритм Шора найти период `omega` функции `f(x)=a^x mod N`. Найдя период, вы получите `a^x mod N=a^(x+omega) mod N` (то есть `a^x mod N=a^xa^omega mod N`), а значит, `a^omega mod N=1`, или, что равносильно, `a^omega-1 mod N=0`. Иными словами, `a^omega-1` кратно `N`, то есть `a^omega-1=kN` для некоторого неизвестного числа `k`.

Если `omega` четно, `a^omega-1` легко разложить как `(a^(omega/2)-1)(a^(omega/2)+1)`, где `a^(omega/2)` является корнем из единицы, поскольку `(a^(omega/2))^2 mod N=1`. Если период `omega` нечетен, повторите алгоритм Шора с другим значением `a`, пока не получите четное число.

Поскольку множители `a^omega-1` содержат простые множители `k` и `N`, эти множители можно найти распределенными между множителями `a^(omega/2)-1` и `a^(omega/2)+1`. Затем можно вычислить наибольший общий делитель (НОД) `a^(omega/2)-1` и `N`, а также `a^(omega/2)+1` и `N`, чтобы получить

нетривиальный множитель N, то есть значение, отличное от 1 и N. Если этого не произошло, например когда a^(omega/2)-1 или a^(omega/2)+1 кратно N, начните атаку заново с другим a.

Получив множители N, вы восстановили закрытый ключ RSA из открытого и можете расшифровывать зашифрованные сообщения или подделывать подписи.

Заметьте, что лучший классический алгоритм факторизации числа N работает за время, экспоненциальное относительно n - битовой длины N (то есть n=log_2N). Однако алгоритм Шора работает за полиномиальное относительно n время, а именно O(n^2(log n)(loglog n)). Это означает, что при наличии квантового компьютера алгоритм Шора дал бы результат за более разумный срок, чем тысячи лет.

Задача дискретного логарифмирования

Задача дискретного логарифмирования требует найти x, имея y=g^x mod p для некоторых известных чисел g и p. Решение этой задачи на классическом компьютере требует (суб)экспоненциального времени, но с помощью эффективного метода нахождения периода алгоритма Шора x можно найти легко.

Например, рассмотрим функцию f(a,b)=g^ay^b. Допустим, требуется найти период этой функции - числа omega и omega', при которых f(a+omega,b+omega')=f(a,b) для любых a и b. Тогда искомое решение x=-omega/omega' mod q, где q - известный параметр, порядок g. Равенство f(a+omega,b+omega')=f(a,b) означает g^omega y^(omega') mod p=1. Подставив вместо y значение g^x, получим g^(omega+xomega') mod p=1, что равносильно omega+xomega' mod q=0, откуда выводится x=-omega/omega'.

Снова общая сложность составляет O(n^2(log n)(loglog n)), где n - битовая длина p. Этот алгоритм обобщается для нахождения дискретных логарифмов в любой конечной коммутативной группе, а не только в группе чисел по модулю простого числа. Поэтому его можно применять и для решения ECDLP - версии задачи дискретного логарифмирования для эллиптических кривых.

Алгоритм Гровера

Другая важная форма квантового ускорения - возможность поиска среди n элементов за время, пропорциональное квадратному корню из n, тогда как любой классический алгоритм потребует времени, пропорционального n. Такое квадратичное ускорение возможно благодаря алгоритму Гровера - квантовому алгоритму, открытому в 1996 году. Я не буду рассматривать внутреннее устройство алгоритма Гровера, поскольку оно по существу представляет собой набор вентилей Адамара, но объясню, какую задачу решает Гровер и как он может повлиять на криптографическую безопасность. Я также покажу, почему симметричный криптографический алгоритм можно спасти от квантовых компьютеров, удвоив размер ключа или хеш-значения, тогда как асимметричные алгоритмы уничтожаются окончательно.

Представьте алгоритм Гровера как способ найти значение x среди n возможных значений, для которого f(x)=1, тогда как для большинства остальных значений f(x)=0. Если условию f(x)=1 удовлетворяют m значений x, Гровер найдет решение за время O(sqrt(n/m)), то есть время, пропорциональное квадратному корню из n, деленного на m. Для сравнения, классический алгоритм не может работать лучше, чем за O(n/m).

Теперь учтем, что f() может быть любой функцией. Например, это может быть условие: «f(x)=1 тогда и только тогда, когда x равен неизвестному секретному ключу K, для которого E(K,P)=C» при

некоторых известных открытом тексте P и шифртексте C, где E() - некоторая функция шифрования. На практике это означает, что при поиске 128-битного ключа AES квантовым компьютером ключ будет найден за время, пропорциональное 2^64, а не 2^128, как при наличии только классических компьютеров. Открытый текст должен быть достаточно длинным, чтобы гарантировать уникальность ключа. (Если открытый текст и шифртекст имеют, скажем, длину 32 бита, множество ключей-кандидатов преобразуют этот открытый текст в данный шифртекст.) Сложность 2^64 намного меньше 2^128, то есть секретный ключ восстановить гораздо легче. Но существует простое решение: чтобы вернуть 128-битную безопасность, достаточно использовать 256-битные ключи! Тогда алгоритм Гровера снизит сложность поиска ключа до 2^(256/2)=2^128 операций.

Алгоритм Гровера также может находить прообразы хеш-функций (см. главу 6). Для нахождения прообраза некоторого значения h определим функцию f() как «f(x)=1 тогда и только тогда, когда Hash(x)=h, иначе f(x)=0». Таким образом, Гровер находит прообразы n-битных хешей ценой порядка 2^(n/2) операций. Как и в случае шифрования, чтобы обеспечить постквантовую безопасность 2^n, используйте вдвое более длинные хеш-значения, поскольку алгоритм Гровера находит прообраз 2n-битного значения не менее чем за 2^n операций.

Итог: симметричные криптографические алгоритмы можно спасти от квантовых компьютеров, удвоив размер ключа или хеш-значения.

> **Примечание.** Существует известный квантовый алгоритм, находящий коллизии хеш-функции за время O(2^(n/3)) вместо O(2^(n/2)), как в классической атаке парадокса дней рождения. Из этого можно заключить, что квантовые компьютеры превосходят классические при поиске коллизий хеш-функций, однако квантовому алгоритму со временем O(2^(n/3)) для работы требуется также пространство, или память, O(2^(n/3)). Предоставьте классическому алгоритму компьютерное пространство O(2^(n/3)), и он сможет выполнить параллельный алгоритм поиска коллизий со временем всего O(2^(n/6)), что намного быстрее квантового алгоритма O(2^(n/3)). (Подробности атаки см. в работе Дэниела Дж. Бернштейна «Cost Analysis of Hash Collisions» («Анализ стоимости коллизий хешей») на cr.yp.to.) Однако в 2017 году криптографы предложили квантовый алгоритм, находящий коллизии за время O(2^(2n/5)) и требующий O(n) квантовой памяти и O(2^(n/5)) классической памяти. Он может превзойти классический поиск (см. eprint.iacr.org).

Почему построить квантовый компьютер так трудно?

Хотя квантовые компьютеры в принципе можно построить, мы не знаем, насколько это трудно и когда это произойдет, если произойдет вообще. По состоянию на середину 2024 года рекорд принадлежит машине с 1121 кубитом (IBM Condor), тогда как для взлома какой-либо криптографии потребовалось бы неделями поддерживать стабильность миллионов кубитов. Иными словами, мы еще далеко от цели.

Трудность построения квантового компьютера связана с необходимостью использовать в роли кубитов чрезвычайно малые объекты - меньше атомов, например фотоны. Из-за столь малого размера кубиты также чрезвычайно хрупки.

Кроме того, для сохранения стабильности кубиты необходимо содержать при чрезвычайно низких температурах (близких к абсолютному нулю). Даже при таких температурах состояние кубитов распадается, и в конце концов они становятся бесполезны. На момент написания книги мы еще не умеем создавать кубиты, сохраняющиеся дольше пары секунд (их времени когерентности).

Еще одна трудность: окружающая среда, например тепло и магнитные поля, может влиять на состояния кубитов и вызывать ошибки вычислений. В теории эти ошибки можно исправить, но сделать это трудно. Для исправления ошибок кубитов нужны квантовые коды исправления ошибок, которые, в свою очередь, требуют множества дополнительных кубитов и достаточно низкой частоты ошибок.

В настоящее время существуют два основных подхода к созданию кубитов: сверхпроводящие схемы и ионные ловушки. Лаборатории Google и IBM отстаивают использование сверхпроводящих схем, при котором кубиты создаются как крошечные электрические цепи, опирающиеся на квантовые явления в сверхпроводящих материалах, где носителями заряда являются пары электронов. Кубиты из сверхпроводящих схем живут очень недолго.

Ионные ловушки, или захваченные ионы, состоят из ионов (заряженных атомов), которыми управляют лазерами, чтобы подготовить кубиты в определенных начальных состояниях. Ионные ловушки обычно стабильнее сверхпроводящих схем, однако работают медленнее и, по-видимому, труднее масштабируются.

Построение квантового компьютера - настоящий проект масштаба полета на Луну. Задача сводится к двум этапам: 1) построить систему с несколькими кубитами, которая стабильна, отказоустойчива и способна применять базовые квантовые вентили; 2) масштабировать эту систему до тысяч или миллионов кубитов, чтобы сделать ее полезной. С чисто физической точки зрения и насколько нам известно, ничто не препятствует созданию больших отказоустойчивых квантовых компьютеров. Но многое возможно в теории и оказывается трудным или слишком дорогим на практике (как безопасные компьютеры). Будущее покажет, кто прав: квантовые оптимисты (предсказывающие большой квантовый компьютер в течение десятилетия) или квантовые скептики (утверждающие, что человечество никогда не увидит квантового компьютера).

Как уже упоминалось, по состоянию на январь 2024 года одним из самых передовых достижений является квантовая микросхема IBM Condor, содержащая 1121 кубит и основанная на сверхпроводящих схемах. Но число кубитов не должно быть единственной мерой при сравнении квантовых вычислительных систем. К другим важным факторам относятся время стабильности, число запутанных друг с другом кубитов и способность надежно исправлять ошибки.

Постквантовые криптографические алгоритмы

Область постквантовой криптографии занимается разработкой алгоритмов с открытым ключом, которые не сможет взломать квантовый компьютер, то есть квантовобезопасных алгоритмов, способных заменить RSA и алгоритмы на эллиптических кривых в будущем, где серийные квантовые компьютеры мгновенно взламывают 4096-битные модули RSA.

Такие алгоритмы не должны опираться на трудную задачу, которая, как известно, эффективно решается алгоритмом Шора, уничтожающим трудность задач факторизации и дискретного логарифмирования. Симметричные алгоритмы, например блочные шифры и хеш-функции, перед лицом квантового компьютера потеряют лишь половину теоретической безопасности, но не будут взломаны столь же катастрофически, как RSA. Они могут стать основой постквантовой схемы.

В следующих разделах мы рассмотрим четыре основных типа постквантовых алгоритмов: основанные на кодах, решетках, многомерных уравнениях и хешах.

Криптография на основе кодов

Постквантовые криптографические алгоритмы на основе кодов опираются на коды исправления ошибок - методы, разработанные для передачи битов по зашумленному каналу. Основная теория кодов исправления ошибок восходит к 1950-м годам. Первая схема шифрования на основе кодов (криптосистема Мак-Элиса) была разработана в 1978 году и до сих пор не взломана. Криптографические схемы на основе кодов можно использовать как для шифрования, так и для подписей. Главное их ограничение - размер открытого ключа, обычно порядка ста килобайтов. Но действительно ли это проблема, если средний размер веб-страницы около 2 Мбайт?

Сначала я объясню, что такое коды исправления ошибок. Допустим, требуется передать последовательность битов как последовательность трехбитных слов, но передача ненадежна, и вы опасаетесь ошибочной передачи одного или нескольких битов: отправляете 010, а получатель принимает 011. Это можно исправить простейшим повторяющим кодом исправления ошибок: вместо 010 передать 000111000 (повторив каждый бит три раза), а получатель декодирует принятое слово, выбрав значение большинства для каждого из трех битов.

Например, получатель декодирует повторяющее кодовое слово 100110111 как 011, потому что 100 содержит два нуля, 110 - две единицы, а 111 - три единицы. Этот конкретный код исправления ошибок позволяет исправить не более одной ошибки в каждом трехбитном фрагменте, поскольку при двух ошибках в одном фрагменте значением большинства окажется неправильное.

Линейные коды - менее тривиальный пример кодов исправления ошибок. В случае линейных кодов кодируемое слово рассматривается как n-битный вектор v, а кодирование состоит в умножении v на матрицу G размером m* n для вычисления кодового слова w=vG. (В этом примере m больше n, то есть кодовое слово длиннее исходного.) Матрицу G можно построить так, чтобы при заданном числе t любая t-битная ошибка в w позволяла получателю восстановить правильное v. Иными словами, t - максимальное число исправимых ошибок.

Для шифрования данных с помощью линейных кодов криптосистема Мак-Элиса строит G как секретную комбинацию трех матриц, а шифрование выполняет, вычисляя w=vG и добавляя некоторое случайное значение e с фиксированным числом битов, равных 1. Здесь G является открытым ключом, а закрытый ключ состоит из матриц A, B и C, таких что G=ABC. Знание A, B и C позволяет надежно декодировать сообщение и восстановить w. Но без этих матриц декодирование слова, а следовательно и расшифрование, должно быть невозможно.

Безопасность схемы шифрования Мак-Элиса основана на трудности декодирования линейного кода при недостатке сведений; известно, что эта задача NP-трудна и потому недоступна квантовым компьютерам. Однако помните: если задача NP-трудна, это еще не означает, что все ее экземпляры невозможно решить на практике. Поэтому необходимо оценивать, какие экземпляры трудной задачи представляет криптосистема и всегда ли они будут трудными. После многолетнего анализа криптографами и специалистами по теории кодирования шифр Мак-Элиса удовлетворяет этому критерию.

Криптография на основе решеток

Решетки - математические структуры, по существу состоящие из множества точек в n-мерном пространстве с некоторой периодической структурой. Например, на рисунке 14-5 показано, как решетку размерности два (n=2) можно представить множеством точек.

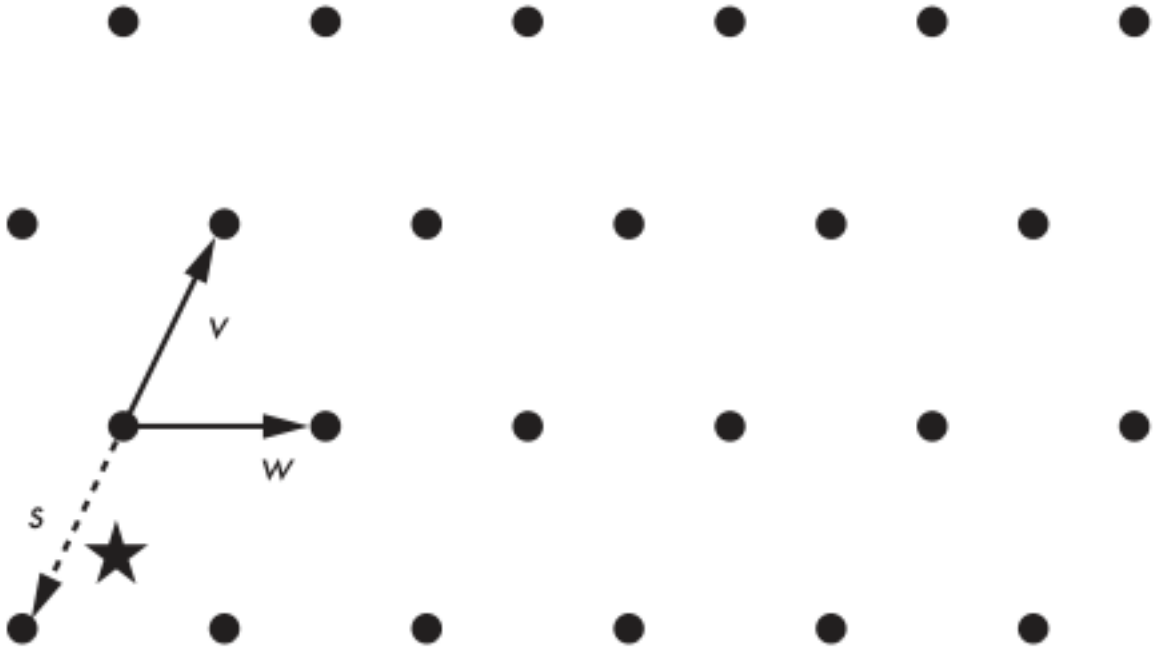

Рисунок 14-5. Точки двумерной решетки, где v и w - базисные векторы решетки, а s - ближайший вектор к точке в форме звезды

Теория решеток привела к появлению обманчиво простых криптографических схем. Я изложу самую суть.

Задача малого целочисленного решения (short integer solution, SIS) - трудная задача криптографии на основе решеток, состоящая в нахождении секретного вектора s из n чисел по заданной паре (A,b), такой что b=As mod q, где A - случайная матрица размером m* n, а q - простое число.

Другая трудная задача криптографии на основе решеток - обучение с ошибками (learning with errors, LWE), состоящая в нахождении секретного вектора s из n чисел по заданной паре (A,b), где b=As+e mod q, причем A - случайная матрица размером m* n, e - случайный вектор шума, а q - простое число. Эта задача очень похожа на декодирование с шумом в криптографии на основе кодов.

SIS и LWE в некотором смысле эквивалентны, и их можно переформулировать как экземпляры задачи ближайшего вектора (closest vector problem, CVP) в решетке, то есть задачи нахождения вектора решетки, ближайшего к заданной точке, путем объединения набора базисных векторов. Пунктирный вектор s на рисунке 14-5 показывает, как найти ближайший к звездообразной точке вектор, объединив базисные векторы v и w.

Считается, что CVP и другие задачи на решетках трудны как для классических, так и для квантовых компьютеров. Но это не переносится непосредственно на безопасные криптосистемы, поскольку некоторые задачи трудны лишь в худшем случае (то есть для самого трудного экземпляра), а не в среднем случае (что и требуется для криптографии). Кроме того, хотя найти точное решение CVP трудно, приближенное решение может находиться значительно легче.

Тем не менее постквантовые криптосистемы на основе решеток показали наилучшее сочетание безопасности и производительности. Стандарты, выбранные NIST в 2022 году, в основном относятся к этому семейству, как вы увидите далее.

Многомерная криптография

Многомерная криптография занимается построением криптографических схем, взломать которые столь же трудно, как решить системы многомерных уравнений, то есть уравнений с несколькими неизвестными, которые перемножаются в уравнениях. Рассмотрим, например, следующую систему уравнений с четырьмя неизвестными x_1, x_2, x_3, x_4:

```
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.
```

Эти уравнения состоят из сумм членов, представляющих собой либо одно неизвестное, например x_4 (члены первой степени), либо произведение двух неизвестных значений, например x_2x_3 (члены второй степени, или квадратичные члены). Чтобы решить систему, необходимо найти значения x_1, x_2, x_3, x_4, удовлетворяющие всем четырем уравнениям. Уравнения могут задаваться над всеми вещественными числами, только целыми числами или конечными множествами чисел. Однако в криптографии они обычно задаются над числами по модулю некоторых простых чисел или над двоичными значениями (0 и 1).

Трудная задача здесь - найти решение случайной квадратичной системы, известная как многомерные квадратичные уравнения (multivariate quadratics, MQ). Эта задача NP-трудна и потому является потенциальной основой постквантовых систем, поскольку квантовые компьютеры не смогут эффективно решать NP-трудные задачи.

К сожалению, построить криптосистему поверх MQ не так просто. Например, если использовать MQ для подписей, закрытый ключ может состоять из трех систем уравнений: L_1, N и L_2. Их объединение в указанном порядке дает другую систему уравнений P - открытый ключ. Последовательное применение преобразований L_1, N и L_2 (то есть преобразование группы значений в соответствии с системой уравнений) эквивалентно применению P, например путем преобразования x_1, x_2, x_3, x_4 в y_1, y_2, y_3, y_4, определенные 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, чтобы L_1 и L_2 были обратимыми линейными преобразованиями (то есть имели уравнения, где члены только складываются, но не умножаются), а N - также обратимой квадратичной системой уравнений. Благодаря этому сочетание трех систем является обратимой квадратичной системой, но ее обратное преобразование трудно определить без знания обратных для L_1, N и L_2.

Тогда вычисление подписи состоит в применении обратных L_1, N и L_2 к некоторому сообщению M, которое рассматривается как последовательность переменных x_1, x_2, ...:

```
S=L_2^(-1)(N^(-1)(L_1^(-1)(M))).
```

Проверка подписи состоит в подтверждении того, что P(S)=M.

Атакующие смогут взломать такую криптосистему, если сумеют вычислить обратное преобразование P или определить L_1, N и L_2 по P. Фактическая трудность решения таких задач зависит от параметров схемы, например числа используемых уравнений, размера и типа чисел. Однако выбирать безопасные параметры трудно, и уже не одна «безопасная» многомерная схема была взломана.

Многомерная криптография не используется в крупных приложениях из-за трудности достижения надежного компромисса между безопасностью и производительностью. Однако практическое преимущество многомерных схем подписи заключается в том, что они создают короткие подписи.

Криптография на основе хешей

В отличие от предыдущих схем, криптография на основе хешей опирается на надежно установленную безопасность криптографических хеш-функций, а не на трудность математических задач. Поскольку квантовые компьютеры не могут взломать хеш-функции, они не могут взломать и то, что опирается на трудность нахождения коллизий или прообразов, - ключевую идею схем подписи на основе хеш-функций.

Криптографические схемы на основе хешей довольно сложны, поэтому рассмотрим их простейший строительный блок - одноразовую подпись Винтерница (Winternitz one-time signature, WOTS), прием, открытый примерно в 1979 году. Здесь «одноразовая» означает, что закрытым ключом можно подписать только одно сообщение, иначе схема подписи становится небезопасной. (WOTS можно сочетать с другими методами, чтобы подписывать несколько сообщений, как вы вскоре увидите.)

Допустим, требуется подписать сообщение, рассматриваемое как число от 0 до w-1, где w - некоторый параметр схемы. Закрытый ключ является случайной строкой K. Чтобы подписать сообщение M, где 0<= M<w, вычислите Hash(Hash(...(Hash(K)))), где хеш-функция Hash применяется M раз. Обозначим это значение Hash^M(K). Открытый ключ равен Hash^w(K), то есть результату w вложенных применений Hash начиная с K.

Подпись WOTS S проверяется подтверждением того, что Hash^(w-M)(S) равна открытому ключу Hash^w(K). Заметьте, что S - это K после M применений Hash, поэтому еще w-M применений Hash дают значение, равное K, хешированному M+(w-M)=w раз, то есть открытый ключ.

У этой схемы есть существенные ограничения:

Атакующие могут подделывать подписи. Из Hash^M(K), подписи M, можно вычислить Hash(Hash^M(K))=Hash^(M+1)(K) - действительную подпись сообщения M+1. Эту проблему можно исправить, подписывая не только M, но и w-M с помощью второго ключа.

Она работает только для коротких сообщений. Если сообщения имеют длину 8 битов, существует до 2^8-1=255 возможных сообщений, поэтому для создания подписи Hash придется вычислять до 255 раз. Для коротких сообщений это может работать, но для более длинных нет: например, подпись 128-битного сообщения 2^128-1 займет вечность. Обходной путь - разделить длинные сообщения на короткие части и подписать каждую независимо.

Она работает только один раз. Если закрытым ключом подписать более одного сообщения, атакующий может восстановить достаточно сведений для подделки подписи. Например, если w=8 и описанным выше приемом для предотвращения тривиальных подделок подписать числа 1 и 7, атакующий получит Hash^1(K) и Hash^7(K') как подпись 1, а также Hash^7(K) и Hash^1(K') как подпись 7. Из этих значений атакующий может вычислить Hash^x(K) и Hash^x(K') для любого x в [1;7] и тем самым подделать подпись от имени владельца K и K'. Простого исправления нет.

Современные схемы на основе хешей используют более сложные версии WOTS в сочетании с древовидными структурами данных и изощренными методами, позволяющими подписывать разные сообщения разными ключами. К сожалению, получаемые схемы создают большие подписи (порядка десятков килобайтов, как SPHINCS+ - один из алгоритмов подписи, выбранных NIST для стандартизации в 2022 году).

Следует также отметить различие между схемами подписи с состоянием и без состояния. SPHINCS+ не имеет состояния, тогда как XMSS имеет состояние, поскольку должна хранить счетчик. Состояние значительно упрощает проектирование алгоритма, но вынуждает пользователей при его применении хранить состояние, например счетчик.

Наконец, конструкции с открытым ключом, опирающиеся только на хеш-функции, могут предоставлять схемы подписи, но не шифрование.

Стандарты NIST

В 2017 году NIST организовал открытый конкурс для выбора подходящих стандартов постквантовых алгоритмов шифрования и подписи. Подобно предыдущим конкурсам, давшим нам AES (Rijndael) и SHA-3 (Keccak), проект NIST по стандартизации постквантовой криптографии предложил криптографам представлять алгоритмы и выполнять криптоанализ алгоритмов других участников, чтобы исключать их из конкурса.

NIST получил 69 заявок, большинство из которых основывалось на решетках. Из них 26 прошли во второй раунд. В июле 2020 года NIST выбрал семь алгоритмов-финалистов и восемь альтернативных алгоритмов. В июле 2022 года NIST объявил первые четыре стандарта:

CRYSTALS-Kyber. Механизм инкапсуляции ключей (key encapsulation mechanism, KEM) на основе решеток - примитив, который можно рассматривать как схему шифрования секретных ключей. Его можно использовать для шифрования данных (в гибридной схеме, где симметричный шифр фактически шифрует данные ключом, зашифрованным KEM) и для согласования ключей подобно протоколам Диффи-Хеллмана.

CRYSTALS-Dilithium. Схема подписи на основе решеток, разработанная той же командой, что CRYSTALS-Kyber.

Falcon. Схема подписи на основе решеток, основанная на несколько иных методах и предположениях, чем Dilithium.

SPHINCS+. Схема подписи на основе хешей, то есть единственный алгоритм не на основе решеток.

По поводу выбора двух схем подписи на основе решеток NIST заявил следующее:

> [Обе] были выбраны за высокую безопасность и превосходную производительность, и NIST ожидает, что они будут хорошо работать в большинстве приложений. Falcon также будет стандартизирован NIST, поскольку могут существовать сценарии, для которых подписи CRYSTALS-Dilithium слишком велики.

Самые короткие подписи Dilithium имеют длину приблизительно 2 Кбайт, а подписи Falcon вдвое короче. NIST также заявил, что стандартизирует SPHINCS+, «чтобы безопасность подписей не опиралась только на решетки».

На момент написания книги проекты стандартов Kyber, Dilithium и SPHINCS+ опубликованы в серии FIPS как FIPS 203 (Module-Lattice-Based Key-Encapsulation Mechanism Standard, «Стандарт механизма инкапсуляции ключей на основе модульных решеток»), FIPS 204 (Module-Lattice-Based Digital Signature Standard, «Стандарт цифровой подписи на основе модульных решеток») и FIPS 205 (Stateless Hash-Based Digital Signature Standard, «Стандарт цифровой подписи без состояния на основе хешей») соответственно, тогда как стандарт Falcon ожидается немного позднее.

Ожидается, что сначала эти постквантовые алгоритмы будут использоваться в гибридных режимах в сочетании с классическим, не квантовоустойчивым алгоритмом для страхования риска слабостей. Например, Kyber обычно используется в сочетании с X25519 - схемой Диффи-Хеллмана на основе Curve25519 - для защиты TLS-соединений.

NIST также объявил четыре алгоритма, перешедшие в «четвертый раунд». Среди них три схемы шифрования на основе кодов: BIKE, Classic McEliece и HQC. Основанная на изогениях SIKE вскоре после объявления оказалась полностью взломана и потому выбыла из конкурса.

Летом 2022 года NIST начал новый проект по поиску дополнительных постквантовых схем подписи, заявив, что «наибольший интерес представляют схемы подписи, не основанные на структурированных решетках», и ожидая заявок с «короткими подписями и быстрой проверкой». В июне 2023 года NIST получил 50 заявок, включая 11 многомерных схем, 7 схем на основе решеток, 6 на основе кодов, 2 на основе хешей и 7 использующих MPC-в-голове. Этот новый метод преобразует протокол многосторонних вычислений (multiparty computation, MPC) в доказательство знания с нулевым разглашением, то есть единый фрагмент данных, проверка которого соответствует проверке подписи. Число и разнообразие предложенных схем приведут к новым атакам и новым методам атак и, будем надеяться, к надежным стандартам.

Что может пойти не так

Постквантовая криптография может быть принципиально надежнее RSA или криптографии на эллиптических кривых, однако она не безошибочна и не всесильна. Наше понимание безопасности постквантовых схем и их реализаций более ограничено, чем обычно, а это несет повышенный риск.

Неясный уровень безопасности

Постквантовые схемы могут казаться обманчиво стойкими, но оказаться небезопасными как против квантовых, так и против классических атак. Иногда проблематичны алгоритмы на основе решеток, например семейство вычислительных задач ring-LWE (варианты задачи LWE, работающие с многочленами). Ring-LWE привлекает криптографов тем, что позволяет строить криптосистемы, взломать которые в принципе столь же трудно, как решить самые трудные экземпляры задач ring-LWE, способные быть NP-трудными. Но если безопасность выглядит слишком хорошей, чтобы быть правдой, обычно так и есть.

Доказательства безопасности зачастую асимптотические, то есть верны лишь для больших значений параметров, например размерности лежащей в основе решетки. Однако на практике используются гораздо меньшие параметры. Даже если схему на основе решеток на вид столь же трудно взломать, как некоторую NP-трудную задачу, количественно оценить ее безопасность по-прежнему сложно. В случае алгоритмов на основе решеток из-за недостаточного понимания этих новых конструкций мы редко имеем ясное представление о лучших атаках на них и стоимости такой атаки в вычислениях или оборудовании. Эта неопределенность затрудняет сравнение схем на основе решеток с лучше изученными конструкциями, например RSA. Однако исследователи продвигаются вперед, и в идеале через несколько лет задачи на решетках будут изучены столь же хорошо, как RSA. (Более технические подробности о задаче ring-LWE см. в превосходном обзоре Пейкерта на eprint.iacr.org.)

Возможное появление больших квантовых компьютеров

Представьте заголовок CNN примерно от 2 апреля 2048 года: «ACME, Inc. раскрыла тайно построенный квантовый компьютер и запускает платформу “взлом криптографии как услуга”». Ладно, RSA и криптография на эллиптических кривых обречены. Что теперь?

Суть в том, что постквантовое шифрование гораздо важнее постквантовых подписей. Сначала рассмотрим подписи. Если вы все еще использовали RSA-PSS или ECDSA как схему подписи, то могли бы выпустить новые подписи с помощью постквантовой схемы и восстановить доверие к ним. Вы отозвали бы старые, небезопасные перед квантовыми компьютерами открытые ключи и

вычислили свежие подписи для каждого ранее подписанного сообщения. После некоторой работы все было бы в порядке.

Паниковать пришлось бы лишь при шифровании данных квантовонебезопасными схемами, такими как RSA-OAEP. В этом случае весь переданный шифртекст мог быть скомпрометирован, поэтому повторно шифровать открытый текст постквантовым алгоритмом бессмысленно: конфиденциальность данных уже утрачена.

А что насчет согласования ключей с Диффи-Хеллманом (DH) и его аналогом на эллиптических кривых (ECDH)? На первый взгляд, ситуация столь же плоха, как с шифрованием: атакующие, собравшие открытые ключи `g^a` и `g^b`, могут с помощью блестящего нового квантового компьютера вычислить секретный показатель `a` или `b` и общий секрет `g^ab`, а затем сформировать из него ключи, которыми был зашифрован трафик. Но на практике Диффи-Хеллман не всегда используется столь примитивно. Фактические сеансовые ключи, шифрующие данные, могут формироваться как из общего секрета Диффи-Хеллмана, так и из некоторого внутреннего состояния системы. Именно так работают современные системы мобильного обмена сообщениями благодаря протоколу, впервые примененному в приложении Signal. Отправляя собеседнику новое сообщение в Signal, система вычисляет новый общий секрет Диффи-Хеллмана и объединяет его с внутренними секретами, зависящими от предыдущих сообщений того же сеанса (который может длиться долго). Такое сложное использование Диффи-Хеллмана значительно затрудняет работу атакующего даже с квантовым компьютером.

Проблемы реализации

На практике постквантовые схемы состоят из кода - программного обеспечения, работающего на некотором физическом процессоре, - а не только абстрактных алгоритмов. Какими бы стойкими ни были алгоритмы на бумаге, они не защищены от ошибок реализации, дефектов программного обеспечения или атак по сторонним каналам. Алгоритм может быть полностью постквантовым в теории, но оказаться взломанным после реализации, например классической компьютерной программой из-за забытой программистом точки с запятой.

Кроме того, такие схемы, как алгоритмы на основе кодов и решеток, в большой степени опираются на математические операции, для ускорения которых в реализации используются разнообразные приемы. Но сложность кода этих алгоритмов одновременно делает реализацию более уязвимой для атак по сторонним каналам, например атак по времени, которые выводят сведения о секретных значениях из измерений времени выполнения. В действительности такие атаки уже применялись к шифрованию на основе кодов (см. eprint.iacr.org) и схемам подписи на основе решеток (см. eprint.iacr.org).

По иронии судьбы, в первое время использование постквантовых схем на практике может оказаться менее безопасным, чем непостквантовых, из-за возможных уязвимостей их реализаций.

Дополнительная литература

Чтобы изучить основы квантовых вычислений, прочитайте классическую книгу Майкла Нильсена и Исаака Чуанга Quantum Computation and Quantum Information (Anniversary Edition) («Квантовые вычисления и квантовая информация», юбилейное издание; Cambridge, 2011). Менее техническая и более увлекательная книга Скотта Аронсона Quantum Computing Since Democritus («Квантовые вычисления со времен Демокрита»; Cambridge, 2013) охватывает не только квантовые вычисления.

Несколько программных симуляторов позволяют экспериментировать с квантовыми вычислениями, например The Quantum Computing Playground по адресу quantumplayground.net или платформа IBM на quantum.ibm.com. Благодаря интуитивно понятной визуализации эти сайты сравнительно просты в использовании.

Последние исследования постквантовой криптографии представлены на pqcrypto.org и связанной с сайтом конференции PQCrypto.

Ближайшие годы обещают быть особенно захватывающими для постквантовой криптографии благодаря продолжению проекта NIST по стандартизации постквантовой криптографии и крупномасштабному развертыванию постквантовых решений.

Глава 15. Криптография криптовалют

Этой главы не было в первом издании книги, вышедшем осенью 2017 года, когда криптовалюты и блокчейн находились на пике ажиотажа. Хотя блокчейн не вполне оправдал обещания перевернуть несколько отраслей, он глубоко повлиял на криптографические исследования и инженерию. Приложения блокчейна принесли новые захватывающие задачи, привлекли свежие таланты и предложили новый способ соединить теорию с практикой.

До того как слово «крипто» стало синонимом криптовалюты, криптографические алгоритмы и протоколы в основном относились к стандартным возможностям, таким как шифрование и защищенные каналы. Более загадочные протоколы оставались уделом узких исследовательских областей и технических статей, обычно отвечали интересам исследователей и представлялись только на академических конференциях. Новые алгоритмы преимущественно создавали академические исследователи, а в реальных условиях они развертывались, если это вообще происходило, не менее чем после пяти лет рецензирования и анализа и появления множества научных работ, описывающих безуспешные попытки криптоанализа.

Блокчейн перевернул этот процесс. Подобно тому как криптографические протоколы Signal и Tor обошли традиционный академический путь, энтузиасты блокчейна были меньше связаны традициями. Новаторские протоколы часто впервые появлялись в публикациях блогов или неформальных белых книгах, а вскоре следовала реализация. Иногда их создатели полностью отказывались от письменных спецификаций, позволяя коду говорить за себя. Академическое сообщество обращало внимание лишь после широкого внедрения. Самый заметный пример - протокол Bitcoin, который перед развертыванием не проходил формального рецензирования.

Многие опытные исследователи обнаружили в области блокчейна новые задачи и часто сотрудничали непосредственно с блокчейн-организациями. Они разработали сложные протоколы, которые не только раздвигали границы и получали признание коллег, но и быстро внедрялись в реальных условиях, затрагивая тысячи, если не миллионы систем. Яркие примеры - эффективные пороговые схемы подписи ECDSA и системы доказательств с нулевым разглашением. В некоторых случаях существовавшие протоколы, почти не имевшие практического применения, нашли значимые сценарии использования, например подписи Боне-Линн-Шахама (BLS).

Эта глава дает обзор таких криптографических алгоритмов и протоколов - как специально созданных для блокчейна, так и расцветших благодаря ему. Я не буду углубляться в определение блокчейна или принципы его работы, поскольку в интернете есть бесчисленные источники. Вместо этого я сосредоточусь на криптографических схемах, лежащих в основе блокчейнов и важных независимо от сценариев их применения. Даже если вы по-прежнему скептически относитесь к феномену блокчейна, надеюсь, эта глава окажется познавательной.

Применения хеширования

Хеш-функции, швейцарские ножи криптографии, имеют в блокчейн-системах множество применений, включая следующие:

Хеширование данных транзакции

Хеш-функции могут создавать дайджест, подписываемый издателем транзакции, обычно с помощью ECDSA-secp256k1 или Ed25519. Проверка подписи гарантирует, что владелец данного адреса одобрил хешированные сведения, после чего транзакция включается в реестр цепочки.

В блокчейне Ethereum хеш транзакции служит ее уникальным идентификатором; например, можно ввести хеш в поле поиска на etherscan.io, чтобы получить связанную транзакцию. Хеши транзакций Ethereum используют Keccak-256 - функцию, похожую на SHA3-256, но не идентичную ей, - и обрабатывают кодированные данные. Эти данные включают адрес получателя, сумму отправленных эфиров (тикер ETH), входные данные смарт-контракта, цену и предел газа, одноразовое значение (увеличивающееся для каждой новой транзакции) и подпись транзакции.

Хеширование содержимого блока

Хеш-функции могут включать дайджест в следующий блок, чтобы «сцеплять» блоки. Например, каждый блок Bitcoin имеет заголовок, включающий хеш предыдущего блока, древовидный хеш записанных транзакций и некоторые метаданные (версию, временную метку, одноразовое значение и так далее). Bitcoin вычисляет хеш предыдущего блока, хешируя его заголовок двойным SHA-256, то есть `SHA-256(SHA-256(заголовок блока))`. Эта необычная конструкция устраняет риск атак с удлинением и создает страховочную сетку на случай обнаружения небезопасности SHA-256, что маловероятно.

Помимо этих базовых сценариев, хеш-функции являются главным строительным блоком ключевых компонентов блокчейн-систем.

Деревья Меркла

Деревья Меркла - разновидность хеш-деревьев, то есть структур данных, вычисляющих корневое значение из значений листьев по древовидному шаблону. В хеш-деревьях родительский узел вычисляется хешированием дочерних узлов. Например, деревья Меркла используются для создания заголовка блока Bitcoin. Они названы в честь специалиста по информатике Ральфа Меркла, чья криптографическая конструкция 1979 года использовала двоичные хеш-деревья.

Вычисление древовидного хеша

Дерево Меркла принимает на вход значения, образующие его листья, то есть значения, по которым вычисляется корень (результат). Древовидные структуры данных обычно изображают с корнем наверху и листьями внизу.

Например, на рисунке 15-1 показано дерево Меркла, хеширующее четыре значения (`A`, `B`, `C`, `D`).

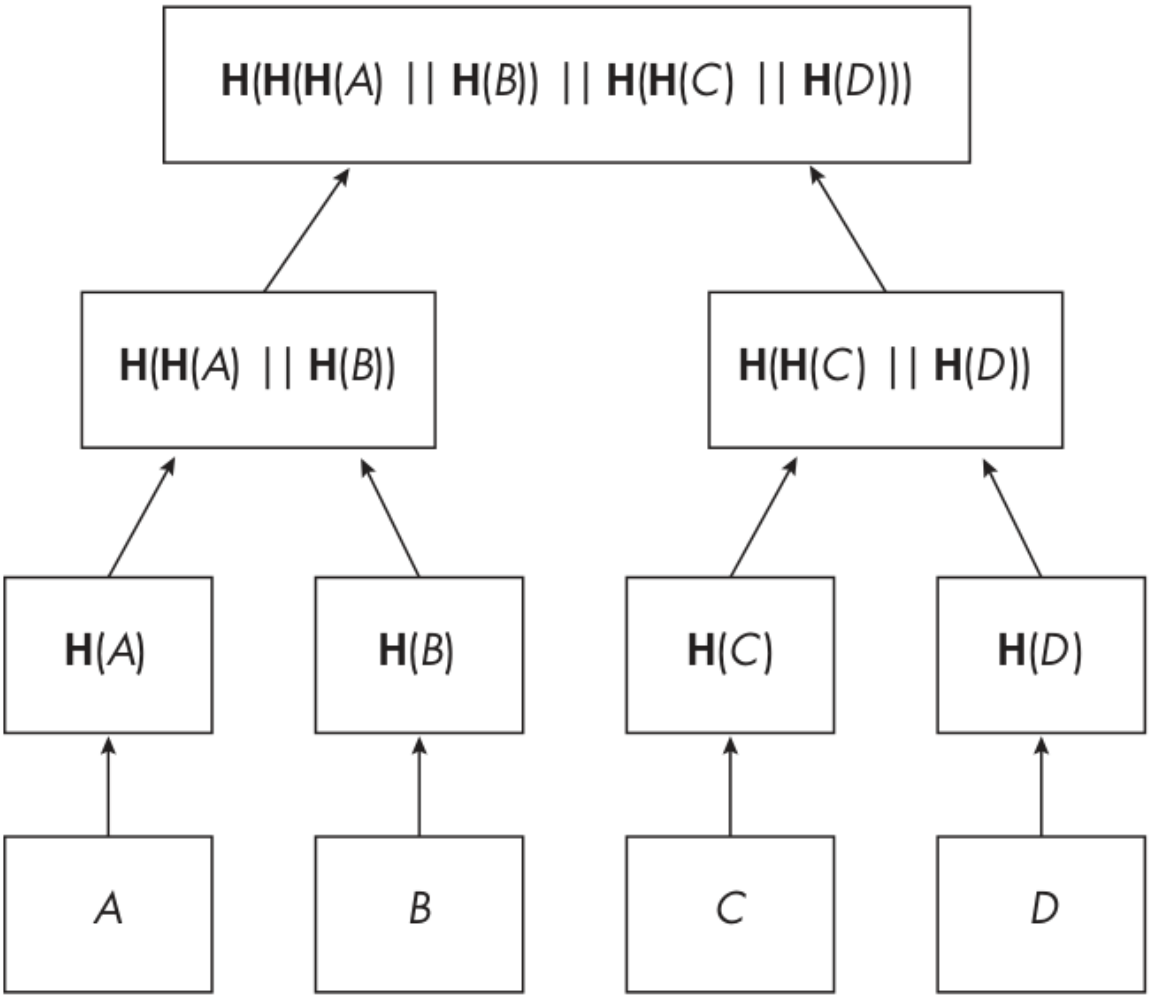

Рисунок 15-1. Хеш-дерево, где листья являются входом, а корень - результатом

Древовидное хеширование выполняется следующим образом:

• Каждое из четырех значений хешируется, давая четыре хеша `H(A)`, `H(B)`, `H(C)` и `H(D)`.
• Каждая последовательная пара хешей хешируется вместе, давая `H(H(A) || H(B))` и `H(H(C) || H(D))`. Здесь хеш-значения сцепляются (что обозначает символ `||`).
• Два хеша хешируются вместе, образуя корень дерева - окончательный результат древовидного хеширования: `H(H(H(A) || H(B)) || H(H(C) || H(D)))`.

Если опустить исходное хеширование входных значений, необходимое только тогда, когда значения еще не имеют размера хеша, четыре значения сводятся к одному деревом из двух уровней, или высоты два. Заметьте: хотя входные данные могут иметь разный размер, все хеш-значения имеют один размер. В общем случае дерево из `n` уровней содержит `2^n` листьев, позволяя обработать до `2^n` значений вычислением `2^n-1` хешей от (хешированных) листьев.

Если число входных значений не равно в точности `2^n` для некоторого целого `n`, распространенный прием состоит в добавлении фиктивных значений (например, равных нулю или последнему значению списка); однако правило дополнения следует выбирать внимательно, чтобы избежать тривиальных коллизий.

Доказательства Меркла

Структура дерева Меркла позволяет доказать, что заданное значение принадлежит списку из 2^n хешированных значений, не вычисляя заново все дерево (что требует порядка 2^n операций), а лишь за время, пропорциональное высоте дерева, то есть числу его уровней n. В зависимости от контекста такое доказательство называется путем принадлежности, доказательством включения или доказательством Меркла. Это убийственная возможность деревьев Меркла, которой нет у хеш-функций общего назначения.

На рисунке 15-2 показано, как это работает. Закрашенные ячейки - значения, достаточные для доказательства того, что V_1 является одним из значений, хешированных для получения корня.

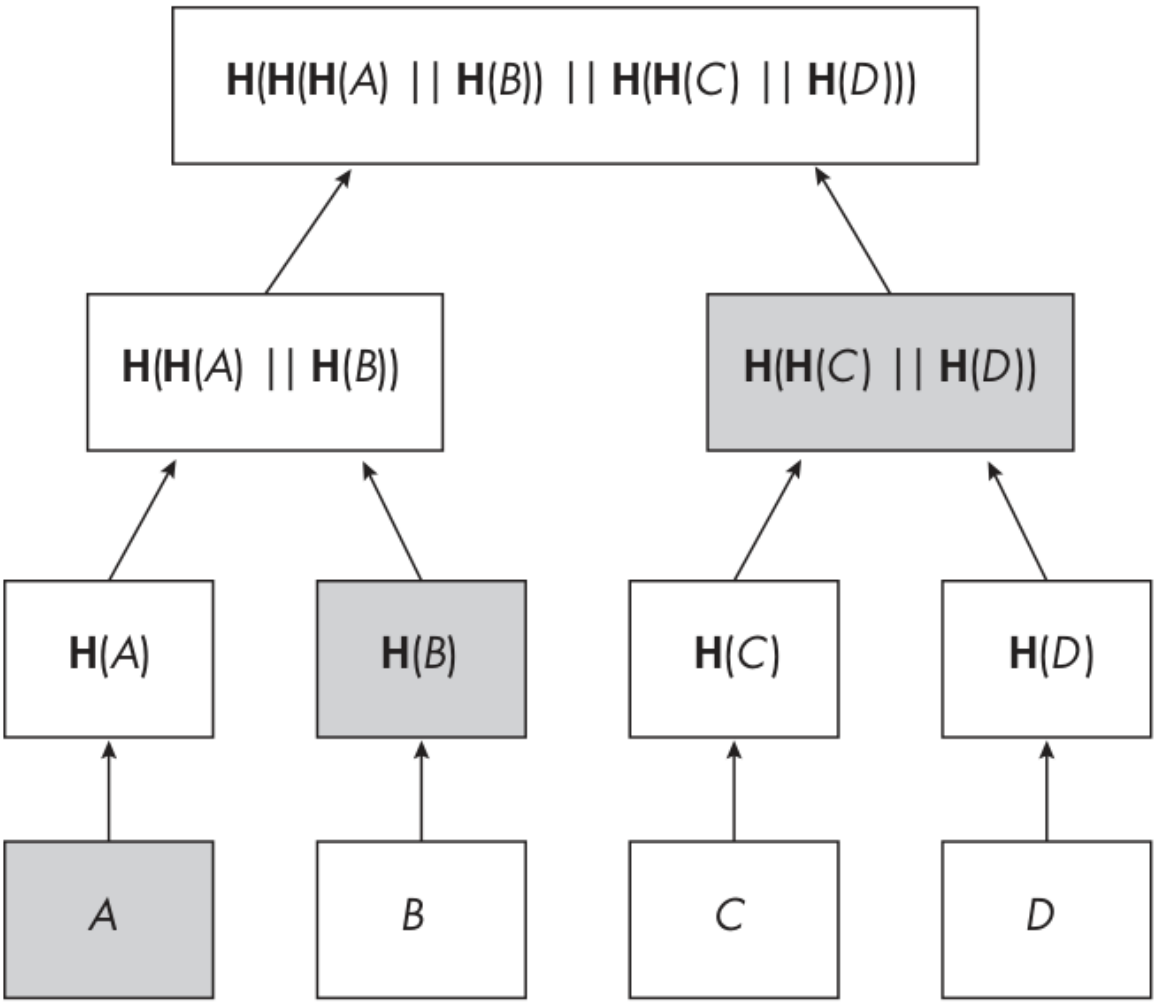

Рисунок 15-2. Дерево Меркла, где закрашенные ячейки образуют путь принадлежности A

Допустим, требуется доказать, что A было одним из хешированных значений, не раскрывая B, C и D. Сначала хешируем A, получая фактический лист хеш-дерева. Затем предположим, что получен путь принадлежности A, состоящий из других закрашенных значений: H(B) и H(H(C) || H(D)). Чтобы проверить принадлежность A дереву, вычислим следующее:

- X=H(H(A) || H(B)), поскольку известны A и H(B).
- H(H(X) || H(H(C) || H(D))), поскольку известно H(H(C) || H(D)).

Для доказательства включения A потребовалось всего два хеширования соседних значений, то есть столько хешированиий, какова высота дерева. Дерево с восемью листьями имеет высоту три, поэтому проверка пути принадлежности требует трех соседних хешей. При 16 листьях нужны

четыре соседних хеша и так далее: n соседних хешей для дерева с 2^n листьями.

Блокчейн-приложения часто используют деревья Меркла, чтобы хешировать транзакции в единый корень Меркла, например включаемый в заголовок блока Bitcoin. Обычный блок Bitcoin регистрирует около 2000 транзакций, для чего требуется дерево высотой 11 (2^11=2048). В этом случае для вычисления корня из листьев нужно 2048-1=2047 соседних хешей, где каждый лист является хешем данных транзакции. Каждый хеш вычисляет двойной SHA-256, поэтому при 2048 транзакциях требуется 2047+2048=4095 двойных SHA-256, или 8190 вызовов SHA-256.

Ethereum использует немного более сложную древовидную структуру данных, объединяющую деревья Меркла с префиксными деревьями Патриции (это не опечатка; английское trie происходит от retrieval) - древовидной структурой, хранящей пары «ключ - значение». Эта структура обслуживает модель состояния Ethereum, значительно отличающуюся от модели UTXO (неизрасходованных выходов транзакций) в Bitcoin, для которой достаточно более простых деревьев Меркла.

Доказательство выполнения работы

Доказательство выполнения работы (proof of work, PoW), вероятно, является важнейшим компонентом протокола консенсуса блокчейна - для блокчейнов, основанных на PoW, а не на доказательстве доли владения (proof of stake, PoS) или других протоколах.

PoW по существу является хеш-функцией, принимающей несколько фиксированных и переменных входов, результат которой для действительности должен соответствовать определенному шаблону. Стороны, стремящиеся решить PoW, многократно вычисляют хеш с разными значениями переменных входов, пока результат не удовлетворит некоторому ограничению.

Например, в Bitcoin и некоторых других блокчейнах на основе PoW ограничение состоит в том, что хеш-значение, рассматриваемое как 256-битное число, должно быть меньше заданного числа. Это также можно представить как требование к хеш-значению иметь заданное число ведущих нулей, если рассматривать хеш как кодировку 256-битного числа от старшего байта к младшему.

Например, в 2022 году наивысшее значение сложности Bitcoin (согласно btc.com) составляло 34 244 331 613 176, или приблизительно 2^45. Умножив его на 2^32, можно найти фактическое значение, меньше которого должен быть хеш: 2^77. Для каждого блока все участники сети (майнеры) совместно выполняют порядка 2^77 вычислений двойного SHA-256, чтобы найти решение PoW. Такое решение состоит из одноразового значения (переменной части входа хеша PoW), которое дает хеш-значение меньше 2^77. Фиксированная часть входа хеша PoW состоит из значений заголовка блока (версии, хеша предыдущего блока, корня дерева Меркла, временной метки и целевого значения сложности).

Если бы PoW не «замедляло» создание блоков в блокчейне, новые действительные блоки можно было бы создавать мгновенно, а значит, истории транзакций можно было бы создавать и пересоздавать по желанию. Достичь окончательности (то есть уверенности, что блок, а следовательно и набор транзакций, после принятия сетью нельзя обратить или изменить) было бы невозможно. В частности, невозможно было бы защитить сеть от двойной траты, то есть расходования одних монет в двух разных транзакциях.

Не все схемы PoW столь просты, как схема Bitcoin, использующая хеш-функцию общего назначения SHA-256. Несколько PoW пытались сделать вычисление на специализированном оборудовании менее эффективным по сравнению с универсальными ЦП, чтобы препятствовать централизации майнинга организациями, инвестирующими в разработку оптимизированной аппаратной технологии майнеров, в отличие от готовых серверов и компьютеров, которыми может

пользоваться любой. Среди реализованных приемов следующие:

Высокая сложность по памяти. Можно заставить PoW использовать большой объем памяти, обычно создавая огромную таблицу и обращаясь к ней по непредсказуемым адресам. Например, PoW Ethereum использовало алгоритм Ethash, требовавший приблизительно 4 Гбайт памяти. (В сентябре 2022 года Ethereum отказался от Ethash и перешел с PoW на механизм доказательства доли владения.)

Виртуальные машины. Подобно некоторым вредоносным программам, можно создать собственный набор компьютерных инструкций, которые приложение виртуальной машины будет переводить в стандартные инструкции, потенциально одновременно используя большой объем памяти для вычисления решения PoW. Такой подход применяет RandomX - алгоритм PoW, принятый блокчейном Monero.

Иерархическое формирование ключей

Пользователи блокчейна обычно хотят управлять несколькими учетными записями, каждая из которых состоит из пары ключей, где:

• Закрытый ключ должен быть секретным, поскольку это ключ подписи, необходимый для подписания транзакций и перевода средств из учетной записи.
• Открытый ключ должен быть открытым, поскольку используется для проверки подписей транзакций. Кроме того, из него формируется адрес учетной записи. Например, Bitcoin формирует адреса из открытого ключа сочетанием хеширования SHA-256 и RIPEMD-160. Заметьте, что в некоторых блокчейнах, например Bitcoin, открытый ключ не публикуется, пока учетная запись не выпустит транзакцию.

Для надежного управления всеми этими ключами разработчики блокчейнов определили иерархические детерминированные кошельки (hierarchical deterministic wallets, HD-кошельки), которые делают управление ключами менее обременительным и рискованным, чем необходимость создавать и резервировать новый случайный ключ для каждой новой учетной записи.

При использовании HD-кошельков генерируется и хранится один секрет - начальное значение (также называемое главным ключом или энтропией). Это единственное случайно сгенерированное значение и единственное значение, дающее энтропию, или неопределенность, а следовательно, секретность формируемым из него закрытым ключам подписи. Приложения кошельков часто кодируют начальное значение как начальную фразу, или мнемонику, - последовательность из 12-24 слов, которую проще сохранить и запомнить.

Рассмотрим работу этого формирования ключей с помощью псевдослучайной функции HMAC-SHA-512 (конструкции HMAC, конкретизированной SHA-512). Из начального значения `S` длиной 128 или 256 битов вычисляется HMAC-SHA-512, где начальное значение служит входом ключа, а идентификатор лежащей в основе эллиптической кривой - входом сообщения (это может быть строка `Bitcoin seed`, `Nist256p1 seed` или `ed25519 seed`). Первые 256 битов результата являются главным ключом `k`, а последние 256 битов - главным кодом цепочки `c`, значением для формирования дополнительных ключей.

Дочерний закрытый ключ можно сформировать из `k` и `c` следующим образом при заданном идентификаторе `i` (числе не больше `2^31`): вычисляется HMAC-SHA-512, где код цепочки `c` служит ключом, а сообщение включает `k` и `i`. 512-битный результат разбирается как 256-битное значение `L`, за которым следует 256-битный код цепочки `R`. В упрощенной записи `L || R=HMAC-SHA-512(c,k || i)`. Затем дочернему закрытому ключу присваивается `k+L`, а его кодом цепочки становится `R`.

В свою очередь, из полученного ключа и кода цепочки можно формировать ключи, создавая иерархию. Например, из ключа с индексом 0 формируются все ключи для Bitcoin, а из ключа с индексом 60 - все ключи для Ethereum. Соглашения, связывающие число с каждой блокчейн-сетью, стандартизированы в документе «SLIP-0044: Registered Coin Types for BIP-0044» («Зарегистрированные типы монет для BIP-0044»).

Если сначала сформировать дочерний ключ с идентификатором 0, а затем из этого ключа и его кода цепочки сформировать ключ с идентификатором, скажем, 29, путь формирования ключа будет `0/29`. Будут выполнены два вызова HMAC-SHA-512, давшие соответственно `L_1` и `L_2` как первые 256 битов результатов, а окончательный закрытый ключ будет равен `k+L_1+L_2`. Поэтому все ключи, сформированные из заданного главного ключа `k`, можно представить как `k` плюс сумма значений, возвращенных HMAC-SHA-512. Такое формирование называется усиленным, поскольку требует и закрытый ключ `k`, и код цепочки `c` родительского ключа.

> **Примечание.** В неусиленном варианте вместо закрытого ключа используется открытый, что позволяет определить открытый ключ дочернего ключа из родительского открытого ключа. Подробнее см. исходный документ стандарта Bitcoin «BIP32: Hierarchical Deterministic Wallets» («Иерархические детерминированные кошельки») и его обобщенный стандарт «SLIP-0010: Universal Private Key Derivation from Master Private Key» («Универсальное формирование закрытых ключей из главного закрытого ключа»).

Алгебраические хеш-функции

Такие хеш-функции, как SHA-3 и BLAKE3, работают с байтами или словами из 4 или 8 байтов, где байт - фрагмент из 8 битов. Входные данные являются последовательностью байтов, каждый из которых может принимать любое из 256 значений от 0x00 до 0xff (255); выходные данные аналогично являются последовательностью произвольных байтов. Это хорошо работает при использовании данных, эффективно преобразуемых в последовательность байтов, и эффективных операций над байтами или словами (таких как XOR, побитовый сдвиг слова и сложение целых чисел).

Математический компьютер

Допустим, имеется компьютер, работающий только с числами в заданном диапазоне, например числами по модулю 13. Операция XOR плохо работает с такими числами, поскольку не все четырехбитные числа меньше 13; например, XOR чисел 10 и 4 дает 14, что выходит за диапазон. Можно привести результат по модулю 13 и получить 1, но тогда `10 XOR 4` даст тот же результат, что `10 XOR 11`, - проблема, которой нет при работе с байтами. Это обычно значительно снижает безопасность хеш-функции, например из-за коллизий.

Хуже того, компьютер знает только операции по модулю 13: сложение, вычитание, умножение и деление. Встроенной инструкции побитового XOR у него нет, поэтому она моделируется арифметическими инструкциями по модулю 13, что непросто и вычислительно неэффективно. Подробности такого моделирования я оставлю в качестве упражнения.

Ваша задача - создать хеш-функцию, которая работает только (или преимущественно) с арифметическими операциями для заданного диапазона чисел и не использует побитовые операции, включая XOR, OR, AND или битовые сдвиги.

Функция такого рода нужна для эффективного выполнения некоторых сложных криптографических протоколов, а именно многосторонних вычислений (MPC) и доказательств с нулевым разглашением, с которыми вы познакомитесь далее в этой главе. Такие протоколы часто работают в области математических структур, например конечных полей (множеств целых чисел по модулю простого числа), и иногда должны «преобразовывать» программу в математические уравнения. В принципе в уравнения можно преобразовать любую программу. Но если программа не оптимизирована для лежащей в основе математической структуры, уравнения становятся очень большими и медленными для вычисления. Алгебраические хеш-функции призваны решить эту проблему: они проектируются одновременно безопасными и легко реализуемыми только арифметическими операциями в конечном поле.

Принципы проектирования

Рассмотрим принципы проектирования Poseidon - хеш-функции, созданной в 2019 году для систем доказательств с нулевым разглашением и быстро принятой множеством блокчейн-систем. Такие доказательства иногда должны выражать хеш-функцию как схему арифметических операций над большим конечным полем, например целых чисел по модулю 255-битного простого числа. В этом случае Poseidon оказался на порядки эффективнее хеш-функций общего назначения вроде SHA-256.

Poseidon использует губчатую конструкцию хеш-функции (см. главу 7), поэтому ему требуется построить перестановку - обратимое преобразование с входом и выходом одного размера. Эта перестановка применяется к состоянию, являющемуся вектором элементов конечного поля. В соответствующих приложениях такое конечное поле обычно состоит из чисел по модулю простого числа длиной от 31 до 381 бита в зависимости от приложения.

Затем, как и большинство хеш-функций, перестановка многократно выполняет ряд раундов, поэтому необходимо спроектировать раундовую функцию.

Наконец, Poseidon делит перестановку на три слоя с тремя разными назначениями, напоминающими раунды AES:

• **Слой уникальности AddRoundConstants**, обозначаемый в документации Poseidon как `ARC()`. Он добавляет постоянные значения к элементам состояния, причем константы различаются для каждого раунда. Уникальность каждого раунда предотвращает атаки, в том числе атаку скольжения. Чтобы не определять и не хранить множество констант, Poseidon генерирует их детерминированным генератором случайных битов, инициализированным кодировкой характеристик экземпляра Poseidon (числа раундов, конечного поля, типа S-блока и так далее).
• **Нелинейный слой, или слой S-блока SubWords**, обозначаемый в документации Poseidon как `S`. Этот слой добавляет конфузию - свойство, при котором входные и выходные значения функции связаны алгебраическими уравнениями высокой степени, то есть максимально далеки от линейных уравнений и уравнений малой степени, которые способен использовать дифференциальный криптоанализ. S-блок преобразует каждый элемент состояния независимо от остальных, обычно отображая элемент поля `x` в `x^3` или `x^5`. Показатель остается сравнительно малым для эффективности вычисления.
• **Линейный слой MixLayer**, обозначаемый в документации Poseidon как `M()`. MixLayer обеспечивает диффузию, то есть распространение различий исходного состояния на все элементы. Например, если состояние состоит из четырехэлементного вектора `(x_1,x_2,x_3,x_4)`, то `M()` заменяет каждый элемент линейной комбинацией всех элементов. Он может заменить `x_1` на результат `2x_1+10x_2+x_3+3x_4`. Такое преобразование соответствует умножению вектора на матрицу. Матрицы Poseidon должны удовлетворять определенным свойствам безопасности и проектироваться для эффективной реализации.

Полный раунд Poseidon применяет три слоя в следующем порядке: `ARC()`, `S` к каждому элементу и `M()`. Частичный раунд применяет `S` только к одному элементу и может использовать другую матрицу в `M()`. Затем экземпляр Poseidon выполняет полные раунды, частичные раунды и снова полные раунды; их число зависит от экземпляра, числа элементов, конечного поля и целевого уровня безопасности.

> **Примечание.** Подробнее о Poseidon см. poseidon-hash.info, исходную статью Poseidon (eprint.iacr.org) и статью об улучшенной конструкции Poseidon2 (eprint.iacr.org).

Poseidon - одна из множества алгебраических хеш-функций, созданных для решения практических задач. Среди других конструкций MiMC, Monolith, Rescue-Prime и Tip5.

Что может пойти не так

Рассмотрим несколько нарушений безопасности, связанных с хеш-функциями и их применениями в мире блокчейна.

Взломанная нестандартная хеш-функция

Блокчейн-проект Iota 2017 года был довольно странным. Он заявлял об использовании архитектуры, отличной от последовательной цепочки блоков большинства блокчейнов и более близкой к ориентированному ациклическому графу (directed acyclic graph, DAG), который был намного менее безопасен. Кроме того, данные кодировались не битами, а тритами - единицами, принимающими три значения вместо двух, - якобы для повышения эффективности вычислений на несуществующих процессорах.

Iota использовал не схему подписи на эллиптических кривых вроде ECDSA или Ed25519, а схему подписи на основе хешей, опиравшуюся на признанную конструкцию Винтерница и тем самым обеспечивавшую постквантовую безопасность. Но Iota также спроектировал собственную нестандартную хеш-функцию Curl.

Iota вошел в десятку самых популярных криптовалют и делал грандиозные заявления о потенциальной полезности и безопасности. Однако нестандартная хеш-функция, которую якобы разрабатывали с помощью искусственного интеллекта, оказалась очень слабой против атак на коллизии. На готовом оборудовании исследователи находили коллизии за считаные минуты, хотя использовать их можно было только в надуманных сценариях атак. Iota быстро исправил хеш-функцию.

После этого фиаско известный специалист по криптографии и безопасности Брюс Шнайер прокомментировал: «В 2017 году оставить криптографический алгоритм уязвимым для дифференциального криптоанализа - ошибка новичка. Это говорит о том, что их систему не анализировал ни один компетентный специалист и что вероятность сделать систему безопасной их исправлением мала».

Кошелек с малой энтропией

Описанная ранее в этом разделе модель иерархического формирования ключей безопасна на бумаге, но на практике - лишь при правильной реализации. Обычно это можно проверить с помощью тестовых векторов из документов стандартов BIP32 и SLIP-0010. Если получаются те же входные и выходные значения, что в документации, реализация, скорее всего, правильна, хотя необязательно безопасна.

В 2022 году разработчики популярного мобильного приложения криптовалютного кошелька Trust Wallet объявили о выпуске версии в виде расширения браузера, использующей технологию WebAssembly (Wasm) для эффективной работы в разных браузерах. Однако Wasm не мог использовать тот же ГПСЧ, что мобильные версии; требовалось определить другой.

Плохой ГПСЧ может быть ахиллесовой пятой криптографии (см. главу 2). В Trust Wallet разработчики использовали ГПСЧ «вихрь Мерсенна» (mt19937), который не является криптографическим ГПСЧ. Его энтропия составляет не более 32 битов, а выходные биты создаются простыми линейными комбинациями значений внутреннего состояния.

Поскольку ГПСЧ Trust Wallet имел 32 бита энтропии, он мог создавать лишь `2^32` разных начальных значений для кошельков пользователей. Атакующий мог вычислить все `2^32` возможных начальных значения и для каждого сформировать закрытые ключи и адреса посредством иерархического формирования ключей. Затем можно было просканировать блокчейны, найти адреса, созданные Trust Wallet, и украсть их токены. Исследователи компании Ledger, обнаружившие недостаток, прокомментировали: «Проведение такой атаки занимает гораздо больше пары часов, но с несколькими ГП ее можно выполнить менее чем за день».

Коллизии из-за отсутствия разделения доменов

Обсудим, как находить коллизии для хеш-функции, устойчивой к коллизиям. Как сказал криптограф Моти Юнг: «Если это звучит невозможно, значит, криптографически интересно».

Представьте следующий простой случай: приложение получает сообщения `A` и `B` от двух сторон, Алисы и Боба, а затем хеширует эти сообщения вместе, чтобы создать уникальное для них хеш-значение. Оно может вычислить `H(A || B)`, хешируя строку, состоящую из `A`, за которым следует `B`. Даже если хеш-функция устойчива к коллизиям, получится одно хеш-значение как для `A=COL` и `B=LISION`, так и для `A=CO` и `B=LLISION`. Возникает хеш-коллизия относительно входных значений приложения, хотя для хеш-функции это не коллизия, поскольку в обоих случаях функция `H` обрабатывает одну строку `COLLISION`.

Чтобы избежать проблемы, кодируйте входные значения приложения в строку, уникальную для каждого входа, без неоднозначной кодировки: по каждой строке должен однозначно определяться исходный набор входных значений. В нашем примере добавление знака доллара (`$`) как разделителя двух входов, по-видимому, устраняет коллизию между `COL || LISION` и `CO || LLISION`: получатся строки `COL$LISION` и `CO$LLISION`, дающие разные хеш-значения.

Но одного символа-разделителя недостаточно для устранения неоднозначности, если этот символ разрешен во входных значениях приложения. Например, рассмотрим строку `COL$$LISION` как сцепление двух входных строк, разделенных знаком `$`. Такая строка могла получиться из двух пар входов: `COL` и `$LISION` либо `COL$` и `LISION`. Заметьте, что проблема не возникает, когда входные значения имеют фиксированный постоянный размер, например первая строка состоит из двух символов, а вторая из трех. Даже в этом случае безопаснее использовать разделитель или кодировку, предотвращающие коллизии, поскольку будущая заплатка может добавить вход переменной длины.

Исследователи находили ошибки этого типа в протоколах пороговой подписи, которые вы увидите далее в этой главе, а также в протоколах электронного голосования, где возникающие хеш-коллизии можно использовать для нарушения свойств безопасности этих протоколов.

Протоколы мультиподписи

В криптографическом протоколе мультиподписи участники совместно создают подпись сообщения, функционально эквивалентную отдельному подписанию сообщения всеми сторонами; полученная подпись означает, что все стороны согласились подписать сообщение. Преимущество в том, что вместо числа подписей, равного числу подписантов, существует только одна. Затем любой, у кого есть открытые ключи всех подписантов, может проверить подпись.

Блокчейн-платформы применяют мультиподписи, когда учетной записью управляют несколько сторон, чтобы гарантировать одобрение выпущенных транзакций всеми сторонами. Их может применять и одна сторона, если у нее несколько ключей на разных устройствах, чтобы единственный скомпрометированный ключ не позволял атакующему выпускать транзакции. Такие протоколы мультиподписи являются одним из многих видов протоколов коллективной подписи, в которых стороны выполняют протокол для создания подписи.

Прежде чем углубляться в технические подробности, уточним отличие этих мультиподписей от родственных протоколов.

Несколько многосторонних подписей

Несмотря на сходное название, рассматриваемые здесь протоколы мультиподписи отличаются от сценариев мультиподписи в цепочке или смарт-контрактов мультиподписи, используемых соответственно в Bitcoin и Ethereum. Последние не являются криптографическими протоколами, поскольку участники независимо отправляют свои индивидуальные подписи в блокчейн-сеть. Сеть, в свою очередь, проверяет правило вида «если транзакция имеет подписи от `pub1` и `pub2`, принять ее» или «если транзакция подписана любыми двумя сторонами из `pub1`, `pub2` и `pub3`, принять ее», а иначе отклоняет транзакцию. Здесь открытые ключи `pub` неявно рассматриваются как идентификаторы сторон, что обычно для блокчейн-протоколов.

По сравнению с мультиподписями в цепочке протоколы мультиподписи создают одну подпись; тогда проверять требуется одну подпись, а не несколько. Сценарии мультиподписи и смарт-контракты вместо этого обрабатывают несколько подписей и состоят из правила проверки, а не протокола на стороне подписантов. В обоих случаях для проверки требуются открытые ключи всех подписантов.

Протоколы мультиподписи также отличаются от двух видов других протоколов коллективной подписи, описанных далее. В обоих протоколах результатом является одна подпись, но различаются способ и исходные данные ее создания.

Протоколы агрегированной подписи

• Как и в мультиподписях, каждый участник имеет собственную пару ключей (открытый и закрытый ключи).
• В отличие от мультиподписей, участники могут подписывать разные сообщения, а не одно. Один участник также может подписывать несколько сообщений.

• Как и в мультиподписях, проверка подписи требует нескольких открытых ключей (или их единой агрегированной версии для поддерживающих это протоколов).

Протоколы пороговой подписи

• В отличие от мультиподписей, участники не используют собственные ключи. Вместо этого у них есть доли (также называемые фрагментами) одного закрытого ключа, и ни один участник никогда не знает ключ полностью, даже во время выполнения протокола.
• Как и в мультиподписях, участники подписывают одно сообщение.
• В отличие от мультиподписей, для проверки требуется один открытый ключ.

Теперь, определив мультиподписи, посмотрим, как они работают в самом известном сценарии применения - подписях Шнорра.

Протоколы подписи Шнорра

Математик Клаус-Петер Шнорр создал названную его именем схему подписи в 1989 году и подал на нее заявку на патент, что препятствовало широкому внедрению до 2008 года, когда схема EdDSA (см. главу 12) была оптимизирована для работы с современными эллиптическими кривыми. Схема Шнорра проще стандарта ECDSA, благодаря чему ее легче превратить в схему мультиподписи. Bitcoin поддерживает подписи Шнорра, добавленные в 2022 году для улучшенной поддержки протоколов мультиподписи.

> **Примечание.** Мы будем использовать аддитивную запись (как в EdDSA и при работе с эллиптической кривой), а не исходную мультипликативную запись (применяемую при работе с целыми числами в мультипликативной группе). Поэтому открытый ключ `A`, связанный с закрытым ключом `a`, является точкой эллиптической кривой `A=aG`, где `G` - заранее заданная базовая точка, а элементы группы являются точками, объединяемыми сложением. В мультипликативной записи вместо этого было бы `A=g^a` для некоторого генератора группы `g`, а элементы группы были бы числами, перемножаемыми друг с другом.

Подписи Шнорра с одним подписантом

Прежде чем рассматривать многостороннее подписание, посмотрим, как работает подпись Шнорра одной стороной. Предположим, у Алисы есть закрытый ключ `a` и связанный с ним открытый ключ `A=aG`. Здесь `a` - число, или скаляр, из заданного диапазона чисел (точнее, конечного поля, над которым определена эллиптическая кривая; обычно это положительные целые числа по модулю некоторого большого простого числа длиной не менее примерно 256 битов), а `G` - фиксированная базовая точка кривой.

Чтобы подписать сообщение `M`, Алиса выполняет следующее:

1. Выбирает секретное случайное число `r` и вычисляет точку `R=rG`. Значение `r` является одноразовым значением - одноразовым закрытым ключом, открытым ключом которого служит `R`.
2. Вычисляет `h=H(R || A || M)` - значение, которое будет «связано» с закрытым ключом `a` и одноразовым закрытым ключом `r` для подписи `M`. Хешируется не только сообщение: через несекретные значения `A` и `R` значение `h` связывается соответственно с подписантом и одноразовым значением. Без этого были бы возможны разные атаки.

3. Вычисляет s=r+ha и возвращает пару (R,s) как подпись. Значение s можно представить как произведение секретного ключа на подписываемые данные, где секрет r маскирует результат; без него закрытый ключ тривиально восстанавливается из подписи.

Подпись проверяется подтверждением того, что sG равно R+H(R || A || M)A. Это работает потому, что из s=r+ha, подставив r+ha вместо s в sG, получаем

```
sG=(r+ha)G=rG+haG=R+hA,
```

где h=H(R || A || M), которое проверяющая сторона должна вычислить из сообщения M, открытого ключа A и части подписи R.

Мультиподписи Шнорра

В мультиподписях имеется не один подписант, а несколько. Для простоты опишем случай с двумя соподписантами: познакомьтесь с Бобом, который будет подписывать сообщения совместно с Алисой. Закрытый ключ Боба равен b, а открытый - B=bG. Для совместного создания мультиподписи сообщения M Алиса и Боб могут действовать следующим образом:

1. Алиса выбирает одноразовое значение r_A, вычисляет R_A=r_AG и отправляет R_A Бобу.
2. Боб выбирает одноразовое значение r_B, вычисляет R_B=r_BG и отправляет R_B Алисе.
3. Они вычисляют R=R_A+R_B и задают h=H(R || A || B || M) - значение, которым Алиса и Боб воспользуются для создания своих частей подписи. Конкретное h связано со сторонами через их открытые ключи (A и B) и связано с текущим сеансом подписания через одноразовое значение R только для конкретного выполнения подписи, определяемого R.
4. Алиса вычисляет s_A=r_A+ha и отправляет его Бобу.
5. Боб вычисляет s_B=r_B+hb и отправляет его Алисе.
6. Оба вычисляют R=R_A+R_B и s=s_A+s_B, а затем возвращают (R,s) как подпись.

Для проверки подписи подтверждают, что sG равно R+h(A+B). Подстановка в sG ранее вычисленного значения s дает

```
sG =(s_A+s_B)G=(r_A+ha+r_B+hb)G
=(r_A+r_B)G+h(a+b)G=R+h(A+B).
```

Заметьте, что проверяющей стороне нужно знать и A, и B, а не только их сумму A+B, поскольку оба значения необходимы для вычисления H(R || A || B || M). Однако если вместо этого определить h как H(R || A+B || M), проверяющая сторона может использовать единый открытый ключ A+B, не зная, что подпись выпущена двумя сторонами. «Слияние» нескольких открытых ключей в один называется агрегацией ключей. Это особенно полезно для сокращения размера хешируемых данных при множестве подписантов.

> **Примечание.** Я описал базовые мультиподписи Шнорра для двух сторон, но протокол масштабируется на произвольное число сторон с открытыми ключами P_1,P_2,...,P_n. В определении замените A+B на P_1+P_2+...+P_n, A || B на P_1 || P_2 || ... || P_n, а «отправляет Бобу/Алисе» на «отправляет всем». Похожий протокол можно применить к EdDSA и Ed25519, вариантам схемы Шнорра.

Что может пойти не так

Протокол мультиподписи Шнорра сравнительно прост, но может дать сбой в следующих сценариях атак.

Атака с компенсацией ключей

В этой атаке Боб убеждает проверяющих подпись в том, что совместно подписал сообщение с Алисой, хотя Алиса не видела сообщения и не взаимодействовала с Бобом. Атакующий может воспользоваться этим в сценарии, где ожидается совместное подписание сообщений Алисой и Бобом, например транзакций, требующих одобрения обеих сторон. В обычном случае проверяющие знают открытый ключ Алисы `A` и открытый ключ Боба `B`, а Боб и Алиса знают ключи друг друга.

Теперь предположим следующее: Алиса отправляет свой открытый ключ `A` всем, включая Боба, но Боб вместо своего открытого ключа `B` передает проверяющим `C=B-A=(b-a)G`, а Алисе - `B`. То, что Боб не знает закрытого ключа, соответствующего `C`, для атаки не имеет значения.

Боб должен подписать сообщение своим закрытым ключом `b`, как в случае одного подписанта, но с `h=H(R || A || C || M)`, словно он подписывает вместе с Алисой. Он возвращает `(R,s)` как подпись, где `R=rG` для выбранного им `r`, а `s=r+hb`.

Ожидая подписи Алисы и Боба, проверяющие подтверждают, что `sG` равно `R+h(A+C)`, и равенство выполняется, поскольку `A+C=A+(B-A)=B`. Таким образом, Боб может подделать мультиподпись, даже не взаимодействуя с Алисой и не зная ее закрытого ключа.

На практике этой атаки можно избежать, потребовав от подписантов доказать знание закрытого ключа, например подписав сообщение. Поскольку Боб не знает закрытый ключ, соответствующий `C`, предоставить такое доказательство он не сможет. Как вы увидите на примере протокола MuSig, атаку можно предотвратить и на уровне протокола.

Атака масштабируется на произвольное число сторон; в таком контексте ее часто называют атакой мошеннического ключа. Получив открытые ключи всех остальных сторон, атакующий Боб просто определит свой открытый ключ как `B-X`, где `X` - сумма открытых ключей всех остальных сторон.

Повторяющиеся одноразовые значения

Как и в ECDSA, повторяющиеся одноразовые значения смертельны для протокола мультиподписи Шнорра. Представьте, что генератор псевдослучайных чисел Алисы дает сбой и дважды создает одно секретное одноразовое значение `r_A` в двух выполнениях протокола: сначала она отправляет Бобу `s_A=r_A+ha`, а затем `s'_A=r_A+h'a`, где первое `h` и второе `h'` зависят также от случайности Боба. Таким образом, `r_A=ha-s_A` и `r_A=h'a-s'_A`, откуда `ha-s_A=h'a-s'_A`, или, что равносильно,

```
a(h-h')=s_A-s'_A,
```

что позволяет вычислить закрытый ключ Алисы `a=(s_A-s'_A)/(h-h')`.

Один из подходов к устранению рисков безопасности из-за отказа случайности - полностью отказаться от случайности. Например, при одном подписанте работает вычисление одноразового значения хешированием сообщения и закрытого ключа, как в Ed25519. Однако задание `r=H(a || M)` не будет работать для мультиподписей: если Алиса и Боб дважды подпишут одно сообщение, Алиса вычислит сначала `s_A=r_A+ha`, а затем `s'_A=r_A+h'a`, причем в обоих случаях `r_A=H(a || M)`, а `h'` будет отличаться от `h`, если вредоносный Боб отправит значение, отличное от `H(b || M)`. Тогда Боб, как и любой подслушивающий обмен данными, снова сможет вычислить `a` как `(s_A-s'_A)/(h-h')`.

Небезопасность параллельного выполнения

Протокол мультиподписи Шнорра небезопасен, когда атакующий может инициировать несколько одновременных протоколов подписи. Атака слишком сложна для описания здесь, но документирована в научных статьях по адресам eprint.iacr.org и eprint.iacr.org.

Более безопасные мультиподписи Шнорра

Чтобы избежать атаки с компенсацией ключей и проблем повторных одноразовых значений, исследователи разработали более сложные протоколы мультиподписи, в частности семейство MuSig: MuSig, MuSig2 и MuSig-DN, где MuSig означает multisignature, а DN - deterministic nonce, детерминированное одноразовое значение. Протоколы MuSig также поддерживают агрегацию ключей, позволяя проверяющей стороне проверять подпись единственным открытым ключом, сформированным из ключей подписантов так, что агрегированный ключ не раскрывает ни число подписантов, ни их открытые ключи.

Посмотрим в действии главный прием MuSig. В простейшем случае двух подписантов, Алисы и Боба, с теми же обозначениями, что в предыдущих разделах, вместо вычисления `s_A=r_A+ha` как своей доли подписи Алиса вычисляет `s_A=r_A+mu_Aha`, то есть умножает часть `ha` на значение `mu_A`. Она вычисляет `mu_A` (где `mu` - греческая буква мю), хешируя список открытых ключей участников, за которым следует ключ Алисы: `H(A || B || A)`. Аналогично Боб вычисляет `s_B=r_B+mu_Bhb`, где `mu_B=H(A || B || B)`: список открытых ключей, за которым следует его ключ.

Затем Алиса вычисляет агрегированный открытый ключ `X=mu_AA+mu_BB` - сумму открытых ключей, умноженных на соответствующие значения `mu`. Хеш сообщения теперь равен `h=H(R || X || M)` вместо `H(R || (A+B) || M)` в уязвимой версии схемы мультиподписи с агрегацией ключей.

Этот прием работает, потому что вредоносный Боб больше не может подделать другой открытый ключ, который «компенсирует» `A` Алисы, как при задании `C=B-A` в атаке с компенсацией ключей. В уравнении `X=mu_AA+mu_BB` Боб должен найти новое значение `B`, которое даст «правильные» коэффициенты `mu`, устраняющие `A` из уравнения. Но теперь это невозможно, поскольку уравнение нелинейно относительно `A` и `B` (линейность часто является синонимом небезопасности - см. главу 2).

Если подписантов больше двух, прием применяется похожим образом: коэффициенты `mu` вычисляются хешированием списка ключей, за которым следует ключ подписанта, а затем открытые ключи агрегируются в один `X` вычислением суммы открытых ключей, умноженных на соответствующие `mu`.

> **Примечание.** Подробнее о протоколах MuSig и безопасном формировании одноразовых значений из сообщения в версии MuSig-DN см. bitcoinops.org.

Протоколы агрегированной подписи

В агрегированных подписях участвуют несколько подписантов, каждый из которых подписывает сообщение (оно может быть разным у всех подписантов), после чего подписи объединяются в одну. Имея только эту подпись, открытые ключи подписантов и подписанные ими сообщения, проверка подтверждает, что все подписанты подписали соответствующие сообщения. Поскольку хранить требуется одну подпись, а не по подписи на каждого подписанта, время проверки пропорционально числу сообщений. Если все подписанты подписывают одно сообщение, проверка может быть столь же быстрой, как проверка подписи одного подписанта, независимо от числа подписантов.

Агрегированные подписи, в частности, используются в Ethereum, а именно на его уровне консенсуса. В этом сценарии узлы-валидаторы одобряют предложения изменить состояние системы (в виде блоков) и применяют агрегированные подписи, чтобы минимизировать место для хранения подписей и время проверки. Они используют схему подписи Боне-Линн-Шахама (BLS), с которой вы познакомитесь в этом разделе, начиная с волшебства, лежащего в основе подписей BLS, - криптографических парингов.

Паринги

В криптографии на эллиптических кривых паринг - это операция, преобразующая две точки из двух групп эллиптических кривых (не обязательно одинаковых) в элемент конечного поля. Паринг точек `P` и `Q` обозначается `e(P,Q)`. Криптографические паринги билинейны, то есть для любых `P`, `Q` и `R`:

```
e(P+R,Q)=e(P,Q)* e(R,Q)

e(P,Q+R)=e(P,Q)* e(P,R).
```

Добавление `R` к одному операнду равносильно умножению результата на паринг `R` с другим. Поэтому

```
e(nP,Q)=e(P,Q)* e(P,Q)*...* e(P,Q)=e(P,Q)^n=e(P,nQ),
```

а для разных точек `P_1,P_2,...,P_n`

```
e(P_1+P_2+...+P_n,Q)=e(P_1,Q)* e(P_2,Q)*...* e(P_n,Q).
```

Внутреннее устройство парингов выходит за рамки книги. Подробности см. в статье Кристин Лаутер и Михаэля Нэрига "Cryptographic Pairings" ("Криптографические паринги", eprint.iacr.org) и в книге Нади эль-Мрабет и Марка Жуа *Guide to Pairing-Based Cryptography* ("Руководство по криптографии на основе парингов", Chapman and Hall/CRC, 2016).

Подписи BLS

Подписи BLS были представлены в 2006 году в статье Дэна Боне, Бена Линна и Ховава Шахама "Short Signatures from the Weil Pairing" ("Короткие подписи из паринга Вейля"). Схема была создана "для систем, где подписи вводятся человеком или передаются по каналу с низкой пропускной способностью". Однако они стали применяться в высокоавтоматизированных системах по высокоскоростным каналам, получившим выгоду от описанной позднее агрегации подписей и открытых ключей.

> **Примечание.** Не путайте подписи BLS с кривыми BLS (Баррето-Линна-Скотта), имеющими с подписями общего автора - Линна. Кривые BLS созданы удобными для парингов и допускают безопасные и эффективные операции. Подписи BLS часто работают с точками на кривой BLS. Например, подпись BLS в Ethereum опирается на BLS12-381.

Подпись и проверка одного подписанта

В подписях BLS закрытый ключ Алисы - скаляр a, а открытый - A=aG для заранее заданной базовой точки G. Чтобы подписать M, она сначала вычисляет H=H(M), где хеш-функция возвращает точку кривой, а не битовую строку или скаляр. Обозначение H следует общему соглашению обозначать точки заглавными буквами. Подпись равна S=aH. Это гораздо проще подписи Шнорра или ECDSA: достаточно хешировать сообщение и умножить результат на закрытый ключ.

Для проверки вычисляют два паринга:

• e(A,H) между открытым ключом и хешем сообщения (где A=aG);
• e(G,S) между базовой точкой и подписью (где S=aH).

Из-за билинейности они равны:

```
e(A,H)=e(aG,H)=e(G,H)^a=e(G,aH)=e(G,S).
```

Если равенство выполняется, подпись принимается, иначе отклоняется. Если не учитывать сложность паринга, подписи на его основе - простейшая схема подписи.

Агрегированные подписи нескольких подписантов

Продолжим использовать волшебство подписей BLS и билинейных парингов и рассмотрим сценарий, в котором n подписантов с закрытыми ключами k_1,k_2,...,k_n и открытыми P_1,P_2,...,P_n подписывают n сообщений M_1,M_2,...,M_n и создают S_1,S_2,...,S_n. Здесь H_i=H(M_i) - хеш i-го сообщения.

Подписи S_i=k_iH_i агрегируются сложением: S=S_1+S_2+...+S_n. По билинейности

```
e(G,S)=e(G,S_1+S_2+...+S_n)=e(G,S_1)* e(G,S_2)*...* e(G,S_n).
```

Поскольку e(nP,Q)=e(P,nQ), каждый e(G,S_i) можно заменить на e(P_i,H_i), "перенеся" множитель k_i в левый операнд: тогда там будет k_iG=P_i, а во втором операнде - H_i вместо S_i=k_iH_i.

После агрегирования проверка сравнивает e(G,S) с произведением всех e(P_i,H_i). Требуется 1+n парингов вместо 2n без агрегирования; без него подписи также занимали бы в n раз больше памяти. Теперь рассмотрим агрегирование и подписей, и открытых ключей.

Агрегированные открытые ключи

Представьте, что все подписанты подписывают одно сообщение M и что все открытые ключи агрегируются в один: P=P_1+P_2+...+P_n. Заметьте, что k_1,k_2,...,k_n по-прежнему обозначают соответствующие закрытые ключи, а H=H(M) - хеш сообщения. Для допустимых подписей S_i получается следующее равенство:

```
e(P,H) =e(P_1+P_2+...+P_n,H)
=e((k_1+k_2+...+k_n)G,H)
=e(G,(k_1+k_2+...+k_n)H)
=e(G,S_1+S_2+...+S_n)
=e(G,S).
```

Таким образом, подписи n сторон для одного сообщения проверяются лишь двумя операциями паринга: e(P,H) и e(G,S). Это чрезвычайно эффективно, поскольку проверка подписи по существу

не зависит от числа сторон: точки можно складывать эффективно, тогда как вычисление парингов дорого. Помимо вычислительной эффективности, агрегирование ключей и подписей также экономит память.

Что может пойти не так

Для безопасности подписи BLS, как и многие другие схемы на эллиптических кривых, должны избегать недопустимых ключей и слабых параметров. Как и предыдущие протоколы, они могут быть уязвимы для атаки с компенсацией ключей. Рассмотрим подробности.

Недопустимые ключи

Подписи BLS описаны в IETF Internet-Draft по адресу github.com. Документ задает KeyGen, CoreSign, CoreVerify и KeyValidate. Последний гарантирует, что открытый ключ "представляет допустимую, не являющуюся нейтральным элементом точку, принадлежащую правильной подгруппе".

Проверка ключа предотвращает использование слабых пар закрытых и открытых ключей, для которых подписи легче подделать. Например, рассмотрим тривиальный случай нулевого секретного ключа `a=0`. Тогда подпись любого сообщения `M` равна `0* H(M)=0`, и потому подпись любого сообщения тривиально подделывается. Открытый ключ тогда равен `0* G=O`, то есть точке на бесконечности. Если проверка ключа отклоняет открытые ключи, равные `O`, она гарантирует, что секретный ключ не равен нулю.

Менее тривиален случай, когда значение точки открытого ключа облегчает создание допустимой подписи без знания закрытого ключа. Не все точки кривой одинаково безопасны, особенно точки малой, а не основной подгруппы.

Если точка открытого ключа принадлежит малой подгруппе, возможных допустимых подписей гораздо меньше, и подделка упрощается. Равным образом, если переданный функции проверки открытый ключ не принадлежит эллиптической кривой, допустимые подписи легко подделать.

Поэтому крайне важно проверять открытые ключи алгоритмом KeyValidate из упомянутой спецификации, как в листинге 15-1.

```
Входы:
- PK, открытый ключ в формате, выдаваемом SkToPk.

Выходы:
- result, либо VALID, либо INVALID

Процедура:
1. xP = pubkey_to_point(PK)
2. Если xP равен INVALID, вернуть INVALID
3. Если xP является нейтральным элементом, вернуть INVALID
4. Если pubkey_subgroup_check(xP) равен INVALID, вернуть INVALID
5. Вернуть VALID
```

Листинг 15-1. Алгоритм KeyValidate гарантирует допустимость открытого ключа BLS

Если вы реализуете подписи BLS, убедитесь, что при проверке ключей и подписей код выполняет все проверки из спецификации BLS.

Атака с компенсацией ключей

В базовой форме агрегированные подписи BLS с агрегированными открытыми ключами подвержены тому же типу атаки с компенсацией ключей, что и подписи Шнорра. Если атакующий знает открытые ключи P_1,P_2,...,P_(n-1) первых n-1 подписантов, он может объявить своим открытым ключом

```
P_n=X-(P_1+P_2+...+P_(n-1)),
```

где X - открытый ключ, для которого он знает закрытый ключ x, так что X=xG. Когда атакующий предъявляет подпись, созданную из x, ничего не подозревающие пользователи проверяют подпись сообщения с открытым ключом P_1+P_2+...+P_n, равным X. Атакующий может единолично подписать сообщение от имени набора предполагаемых подписантов.

Чтобы предотвратить эту атаку, пользователи могут доказать знание закрытого ключа для своего открытого ключа, подписав сообщение. Поскольку атакующий не знает закрытый ключ для P_n, он не пройдет такую проверку.

Другая мера защиты - изменить схему агрегированной подписи так, чтобы возвращаемая подпись была не просто суммой S=S_1+S_2+...+S_n, а суммой с коэффициентами, сформированными из открытых ключей:

```
S=t_1S_1+t_2S_2+...+t_nS_n.
```

где t_i=H(P_i || P_1 || P_2 || ... || P_n) для i=1,2,...,n. Тогда агрегированный открытый ключ, используемый для проверки подписи, равен P=t_1P_1+t_2P_2+...+t_nP_n. Убедиться, что этот прием работает, можно, проверив, что e(P,H(M)) по-прежнему равно e(G,S).

Протоколы пороговой подписи

Пороговые подписи отличаются от мультиподписей и агрегированных подписей, требующих, чтобы каждый участник протокола подписания имел собственные открытый и закрытый ключи, тем, что здесь существует один закрытый ключ k, один открытый ключ P и n участников, каждый из которых имеет отдельную долю ключа k_i. Параметр t (порог) определяется так, что t<n, и выполняются следующие условия:

• t+1 подписантов могут совместно выпустить допустимую подпись некоторого сообщения, проверяемую с помощью открытого ключа P, причем ни один подписант не узнает закрытый ключ k. Для этого выполняется протокол, использующий доли k_i каждого подписанта и подписываемое сообщение, но никогда не раскрывающий закрытый ключ ни одной стороне.
• Набор из t подписантов или меньшего числа не может создать подпись и, следовательно, также не может определить закрытый ключ k.

Выпущенная подпись выглядит как обычная подпись одного подписанта и так же проверяется.

Пороговые подписи - особый вид многосторонних вычислений (multiparty computation, MPC), класса протоколов, в которых n участников вычисляют результат некоторой функции f(x_1,x_2,...,x_n) так, что каждый участник знает свой соответствующий вход x_i и узнает результат функции, но не узнает входы x_i других участников. В случае пороговых подписей значения x_i являются долями закрытого ключа, а результатом служит подпись. Подробнее о том, что такое доля, я расскажу в разделе «Методы разделения секрета».

Преимущество пороговых подписей состоит в «сокрытии» числа соподписантов и их личностей, поскольку проверяющие видят лишь подпись от одного закрытого ключа. Хотя некоторые схемы мультиподписи и агрегированной подписи обладают тем же свойством, пороговые подписи лучше

подходят для сценария доверительного хранения криптоактивов, потому что существует только один закрытый и один открытый ключ; пороговые подписи можно непосредственно применить для распределения контроля над любым адресом.

Варианты использования

Пороговые подписи используются в управлении криптовалютой и цифровыми активами для распределения контроля над адресом между несколькими системами или сторонами. Они могут служить для разделения контроля над учетной записью между поставщиком услуг и устройством пользователя: каждая сторона имеет одну долю ключа, и для подписания транзакции, а значит, расходования средств, они должны совместно выполнить протокол. Такая конфигурация гарантирует, что атакующий не сможет самостоятельно авторизовать транзакции, даже если взломает устройство пользователя и получит доли ключа. Аналогично поставщик не может инициировать транзакции без согласия пользователя. Однако надежная реализация этой модели сопряжена с трудностями, особенно в аспектах управления ключами (генерации ключей, ротации ключей, резервного копирования и так далее).

Организация также может применять пороговые подписи, чтобы распределить хранение средств между системами: разными типами устройств, центрами обработки данных, операционными системами и программными компонентами. Это особенно уместно для холодных кошельков и счетов с крупными активами. Одних пороговых подписей недостаточно: нужны всеобъемлющие меры и средства контроля. Доступ к ИТ-компонентам нужно разделить так, чтобы разные люди или поставщики ИТ-услуг имели доступ к разным системам, а значит, к разным долям. Кроме того, инициирование и одобрение транзакций должны быть под строгим контролем и сопровождаться аудиторским следом.

Модель безопасности

Как и для всех криптографических протоколов, мы должны определить, что значит безопасность пороговой схемы подписи. Модель включает цель безопасности (что атакующему должно быть трудно сделать) и модель атакующего (допущения о его возможностях). Рассмотрим обе характеристики.

Цели безопасности

Главная цель та же, что и у подписи одной стороны: атакующий не должен суметь подделать допустимую подпись, а значит, не должен суметь определить закрытый ключ. Из этого также следует конфиденциальность входов: доли ключа не должны утекать к другим сторонам. Наконец, протокол должен обеспечивать корректность: вычисленная подпись должна быть допустимой и доступной всем участникам выполнения.

Модели атакующего

Единой модели безопасности для пороговых подписей нет: атакующий характеризуется по нескольким измерениям. Во всех сценариях он может активно атаковать сетевые коммуникации: перехватывать, изменять и внедрять сообщения. Защищенный канал с аутентификацией и шифрованием пресекает эти атаки. Также предполагается надежная связь: все сообщения получаются в порядке отправки.

Модель рассматривает коррупцию участников: атакующий компрометирует их системы, узнает секреты и подчиняет их своей воле. Предполагается, что он не может коррумпировать более `t` сторон, иначе по самому определению порога сможет подделывать подписи.

Рассмотрим параметры, которые нужно учитывать, когда атакующий способен коррумпировать участников.

Сначала атакующего характеризуют по числу сторон, которые он может коррумпировать. Существуют две категории пороговых подписей, каждая из которых определяется максимальным числом злонамеренных участников, которых можно компрометировать без нарушения безопасности.

Честное большинство. Атакующий может коррумпировать менее половины сторон, поэтому `t<n/2`. Протокол `(t,n)` по определению должен оставаться безопасным при `t` скомпрометированных участниках. Такие протоколы обычно эффективнее, но не допускают произвольных `t`: например, `(4,5)` невозможен в этой модели, так как требует переносить до четырех коррумпированных сторон.

Нечестное большинство. Модель допускает любое `t` от 1 до `n-1`. Можно создать протокол, где скомпрометированы все из `n` сторон, кроме одной, но злонамеренные стороны все еще не могут подделать подпись или восстановить закрытый ключ. Модель гибче по порогу, но часто требует более сложных и надежных механизмов.

Атакующего также характеризуют по тому, что он может делать после коррупции стороны и получения ее секретных значений, включая долю ключа. Это определяют две модели атакующего:

Пассивный, или честный, но любопытный. Он узнает информацию от скомпрометированных сторон, но не может заставить их отклониться от протокола. Это модель компрометации "только для чтения", где атакующий получает снимок памяти системы - и хранилища, и энергозависимой памяти (ОЗУ, регистры процессора).

Активный, или злонамеренный. Стороны могут произвольно отклоняться от предписанного протокола. Это модель полностью захваченной атакующим системы или злонамеренного инсайдера - оператора, администратора или облачного поставщика.

Протокол, безопасный от активных атакующих, всегда безопасен от пассивных, но не наоборот.

Атакующий может выбирать стороны для коррупции двумя способами, которые задаются следующими моделями типа коррупции:

Статическая коррупция. Атакующий должен выбрать участников до начала протокола.

Адаптивная коррупция. Атакующий может дождаться начала, а затем выбрать участников и узнать всю историю их операций в протоколе.

Протокол, безопасный от статической коррупции, всегда безопасен от адаптивной, но не наоборот. Однако существуют приемы преобразования статически безопасного протокола в адаптивно безопасный.

Методы разделения секрета

Ключевой составляющей пороговых подписей служат протоколы разделения секрета: они разбивают секрет на доли, распределенные между сторонами, так, чтобы стороны могли совместно восстановить исходный секрет. Их, в частности, можно применять для резервных копий закрытых ключей, где разные стороны хранят разные доли в разных местах.

Аддитивное разделение

Простейший способ разделить секретное число `s` - представить его `n` значениями `s_1,s_2,...,s_n`, такими что `s_1+s_2+...+s_n=s`. Например, по модулю 100 число `s=47` можно случайно разделить на четыре доли: выбрать три случайных числа от 0 до 99, скажем `s_1=12`, `s_2=94`, `s_3=80`, и задать `s_4=s-s_1-s_2-s_3=61`. Вычитание выполняется по модулю 100, так что `-1=99`, `-2=98` и так далее.

Этот подход очень прост, но для восстановления секрета нужны все доли.

Пороговое разделение

Пороговое разделение ближе к пороговым подписям: для `n`, `t<n` и `s` оно создает доли так, что секрет можно восстановить из любых `t` из `n` долей.

Известнейший метод - разделение секрета Шамира. Он использует свойство: для многочлена степени `t`

```
f(x)=a_0+a_1x+a_2x^2+a_3x^3+...+a_tx^t
```

достаточно `t+1` значений `f(x)` в `t+1` различных точках `x`, чтобы определить все фиксированные коэффициенты `a_i`.

Для порогового разделения задают `a_0=s`, выбирают случайные `a_1,...,a_t` и вычисляют `f(x)` для `n` различных `x`: это и будут `n` долей.

Восстановление коэффициентов из `f(x)` называется интерполяцией Лагранжа - по имени итальянского математика XVIII века, разработавшего общий метод определения уравнения кривой по списку точек. Геометрически уравнение степени 1, `a_0+a_1x`, задает прямую, которой достаточно двух точек. Уравнение степени 2, `a_0+a_1x+a_2x^2`, задает параболу, которой достаточно трех точек.

В реализациях часто есть функция `Lagrange()`, вычисляющая интерполяцию и возвращающая `a_0`, то есть общий секрет. Если для его восстановления нужно объединить значения `f(x)`, можно определить `Lagrange(s_1,s_2,...,s_t)`, возвращающую `s`.

Это работает для любой комбинации `t` различных долей, а не обязательно первых `t`. Подробности `Lagrange()` слишком техничны для этой книги, но функция реализуется как ряд базовых сложений, умножений и обратных элементов.

Тривиальный случай

Один из простейших видов пороговой подписи использует подпись BLS, которую мы ранее рассматривали в контексте агрегированных подписей. Напомню: для закрытого ключа k подпись BLS вычисляется умножением k на точку кривой H=H(M). С помощью аддитивного разделения можно создать пороговую схему с параметрами (n-1,n). Например, при n=3 ключ разбивают на три доли так, что k_1+k_2+k_3=k. Затем каждая сторона вычисляет свою долю подписи, умножая свою долю k_i на H. Сложение трех долей дает следующее:

```
k_1H+k_2H+k_3H=(k_1+k_2+k_3)H=kH.
```

Сложение трех долей дает допустимую подпись ключом k, хотя ни одна сторона не знает k. Сложив свои доли, стороны могли бы восстановить k, но две доли не раскрывают о нем ничего, если доли созданы случайно.

Чтобы создать схему пороговой подписи с произвольными t и n, примените метод разделения секрета Шамира и воспользуйтесь линейностью операции Lagrange(): создайте доли ключа k_i, как описано в разделе «Пороговое разделение», и вычислите подпись следующим образом:

```
Lagrange(k_1H,k_2H,k_3H)=Lagrange(k_1,k_2,k_3)H=kH.
```

Снова получается допустимая подпись ключом k.

Простой случай

Теперь вычислим подписи Шнорра в пороговой конфигурации, что немного техничнее, чем для подписей BLS. Напомню: схема Шнорра вычисляет подпись как s=r+ha, где h=H(R || A || M), r - случайное одноразовое значение для конкретной подписи, а a - закрытый ключ подписанта. Подпись также содержит R=rG, открытое значение одноразового значения. Благодаря линейности по секретным значениям эту схему сравнительно легко превратить в схему пороговой подписи: заметьте, что секрет r прибавляется к секрету a, умноженному на несекретный хеш h.

Представьте простейший случай двух подписантов с аддитивным разделением закрытого ключа подписи k=a+b, где a и b - соответствующие доли ключа двух подписантов. Для подписания обе стороны могут создать секретные одноразовые значения r_A и r_B с соответствующими открытыми значениями R_A=r_AG и R_B=r_BG. Затем стороны обмениваются этими значениями и вычисляют открытое одноразовое значение R=R_A+R_B, которое войдет в подпись. Таким образом, R является открытым значением, сформированным из закрытого значения r=r_A+r_B, поскольку

```
R_A+R_B=r_AG+r_BG=(r_A+r_B)G.
```

Затем стороны вычисляют доли подписи s_A=r_A+ha и s_B=r_B+hb, сумма которых равна

```
(r_A+ha)+(r_B+hb)=(r_A+r_B)+h(a+b)=r+hk.
```

Получается подпись r+hk ключом k, хотя в вычислениях участники использовали только a и b. Для произвольных t и n вместо аддитивного разделения применяют схему Шамира.

Предыдущая конструкция недостаточно безопасна, чтобы удовлетворить всем требованиям безопасности схемы пороговой подписи. В частности, она уязвима для атак с компенсацией ключей, если участники не фиксируют свои одноразовые значения на предварительной фазе протокола, например отправляя хеш своего одноразового значения. У нее есть и ряд более тонких уязвимостей, устраненных в протоколе подписей Flexible Round-Optimized Schnorr Threshold (FROST), созданном криптографами Челси Комло и Яном Гольдбергом в 2020 году и описанном по адресу eprint.iacr.org.

> **Примечание.** Протокол EdDSA из главы 12 похож на подписи Шнорра, но вычисляет одноразовое значение как хеш сообщения, а не выбирает его произвольно и случайно. Это усложняет создание пороговых протоколов, совместимых с исходной спецификацией EdDSA.

Трудный случай

Схема подписи, которую труднее всего выполнять в пороговой конфигурации, одновременно является самой распространенной. Алгоритм подписи ECDSA (см. главу 12) сложнее подписей Шнорра и EdDSA, потому что включает деление. Для хеша сообщения `h=H(M)` подписант выбирает случайное число `k`, вычисляет число `r` из координат точки `kG` и вычисляет подпись как `s=(h+ra)/k`, где `a` - закрытый ключ подписанта.

Эффективные и безопасные пороговые подписи ECDSA остаются сложной исследовательской задачей среди криптографов. Первые практические протоколы появились в конце 2010-х годов благодаря сценарию применения в криптовалютах: большинство ведущих криптовалют, включая Bitcoin и Ethereum, тогда поддерживали в качестве схемы подписи транзакций только ECDSA.

Криптографы разработали несколько подходов к построению протоколов пороговой подписи ECDSA. Например, протокол Йехуды Линделла 2017 года "Fast Secure Two-Party ECDSA Signing" ("Быстрое безопасное двустороннее подписание ECDSA", eprint.iacr.org) требовал схему фиксации, схему гомоморфного шифрования и систему доказательств с нулевым разглашением. Сложность таких протоколов затрудняла их понимание, реализацию и анализ безопасности, что привело к ряду изъянов безопасности в развернутых системах.

Что может пойти не так

Конкретные проблемы безопасности пороговых протоколов часто сложны и затрагивают детали конструкций, не рассматривавшиеся в этой книге. Поэтому вместо отдельных проблем я обсужу их категории.

Эти категории проблем затронули реальные внедрения пороговых подписей - от широко применяемого ПО с открытым кодом до коммерческих решений.

Статьи и код

Когда инженерам нужно реализовать протокол пороговой подписи по посвященным ему научным статьям, они сталкиваются с трудностями. Эти статьи, предназначенные прежде всего для исследователей криптографии, часто различаются по редакционному качеству, сложны, насыщены математикой и довольно новы. Эта новизна может означать, что экспериментальный протокол не так безопасен, как предполагалось. Эти факторы порождают целый ряд проблем безопасности в реальных условиях, которые можно разделить на четыре основные области:

Небезопасный протокол. Если протокол не безопасен на бумаге, он не станет безопаснее в реализации. К распространенным проблемам относятся упущенные краевые случаи или недостаточная проверка при получении входных данных от других сторон. Например, некоторые протоколы не проверяли, попадает ли зашифрованное число в ожидаемый диапазон, что приводило к практическим атакам.

Неполное описание. Научные статьи не являются техническими спецификациями, а пишутся для академической аудитории, поэтому в них часто отсутствуют практические подробности реализации, например организация сети и кодирование. Примечательный пример - атака TSSHOCK на пороговые подписи, использовавшая неоднозначное кодирование элементов входа хеш-функции, как описано в разделе «Коллизии из-за отсутствия разделения доменов».

Неполная реализация. Сложность протоколов пороговой подписи с их многочисленными подкомпонентами и подробными требованиями может привести к тому, что проверки безопасности будут упущены, особенно если они упомянуты только в приложениях. Например, один протокол требовал посредством доказательства с нулевым разглашением проверить, что число `N=pq` для шифрования Пайе является произведением двух достаточно больших простых чисел, однако в реализации эту проверку пропустили, допустив небезопасные значения `N`.

Небезопасный выбор компонентов. В описаниях протоколов обычно не говорится «используйте хеш-функцию BLAKE3» или «используйте 256-битную эллиптическую кривую nistp256»; вместо этого говорится «используйте хеш-функцию и эллиптическую кривую, обеспечивающие необходимый уровень безопасности». Поэтому реализаторы должны выбрать подходящие примитивы и безопасно использовать их программные интерфейсы. Например, модуля RSA длиной 1024 бита недостаточно для обеспечения 128-битной безопасности.

Риски также возникают, когда реализаторы сознательно меняют протокол ради эффективности или конкретного сценария. Обычно это кончается плохо. Примеры см. в "Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations" Дмитро Тимоханова и Омера Шломовица (eprint.iacr.org) и "Practical Key-Extraction Attacks in Leading MPC Wallets" ("Практические атаки с извлечением ключей на ведущие MPC-кошельки") Николаоса Макриянниса, Орена Йомтова и Арика Галански (eprint.iacr.org).

Аспекты управления ключами

Как-то друг заметил: «На каждые 10 строк кода шифрования приходится 1500 строк управления ключами». Это подчеркивает критическую важность и сложность процессов управления ключами, включая создание, хранение, резервное копирование и восстановление ключей. Хотя эти аспекты могут упускаться в академических статьях, сосредоточенных на теории, в практических приложениях они жизненно важны. Инженеры и специалисты по безопасности, внедряющие пороговое подписание в промышленной среде, должны отдавать этим вопросам управления ключами высший приоритет: даже самые надежные протоколы пороговой подписи не заменяют всеобъемлющие методы управления ключами.

Рассмотрим основные аспекты управления ключами для организации, применяющей пороговые подписи для защиты значительных криптовалютных активов:

Генерация ключей. Независимо от того, используется распределенная или централизованная генерация ключей, необходимо гарантировать, что секретные значения не раскрываются неуполномоченным системам или сторонам ни во время, ни после генерации ключей. Обычно такая гарантия обеспечивается строгими процессами, например церемониями ключей, обеспечением целостности цепочки поставок и аудиторского следа, которые позволяют показать, что ключи созданы правильно. Не позволяйте другой стороне, например облачному поставщику, генерировать ключи от вашего имени или иметь возможность прочитать их в любой момент.

Хранение ключей. Хранение ключа в виде нескольких долей вместо единого значения не уменьшает потребности в безопасном хранении. Защита многочисленных секретов на разных платформах может оказаться более сложной задачей, чем защита одного секрета на единой платформе. Доли ключа должны храниться в некотором виде защищенной памяти и быть

защищены от несанкционированного доступа, подделки и физических атак.

Резервное копирование и восстановление ключей. Реализация пороговой схемы, в которой, например, для подписания транзакции требуются три из шести долей, защищает от потери долей ключа или отказов систем. Тем не менее это не отменяет потребности в резервных копиях ключа, которые также следует хранить как пороговые доли. Крайне важно распределить доступ к этим резервным долям между разными сторонами. Кроме того, периодически проверяйте надежность резервных копий, чтобы убедиться, что они не скомпрометированы и могут эффективно использоваться для восстановления ключа при необходимости, например во время учений по аварийному восстановлению.

Доказательства с нулевым разглашением

Доказательства с нулевым разглашением (ZKP) - один из самых мощных инструментов криптографов. Это протоколы двух сторон, доказывающей и проверяющей, где первая убеждает вторую в истинности чего-либо, не раскрывая ничего о том, почему это истина. Например, доказывающая сторона может доказать, что знает решение трудной вычислительной задачи, не раскрывая самого решения. Можно создать ZKP для любой NP-полной задачи, чтобы доказать знание решения, не раскрывая его.

В более общем виде ZKP используются для доказательства утверждений наподобие "зашифрованное в этом шифртексте число лежит между 100 и 200" или "для данных открытого текста `P` и шифртекста `C` я знаю секретный ключ `K`, для которого `C=AES(K,P)`".

Для нетехнического знакомства с ZKP я рекомендую видео криптографа Амита Сахаи "Computer Scientist Explains One Concept in 5 Levels of Difficulty" ("Ученый-информатик объясняет одно понятие на пяти уровнях сложности", youtu.be). Математические подробности см. по ссылкам на github.com.

> **Примечание.** Термин "доказательство с нулевым разглашением" упрощает более точные термины исследовательской литературы. Многие такие "доказательства" в действительности являются аргументами знания с нулевым разглашением. Термин "доказательство" исследователи сберегают для безусловной безопасности, а "аргумент" - для вычислительной. Кроме того, термин "свидетель" обозначает секрет, позволяющий доказывающей стороне доказать свое утверждение; из-за более широкой области он обобщает секрет или секретный ключ.

Модель безопасности

Что значит безопасность ZKP? Кто атакующий - доказывающая или проверяющая сторона? Как она может атаковать протокол? Ответим, рассмотрев цели безопасности и модели атакующего.

Цели безопасности

Безопасное ZKP должно удовлетворять следующим понятиям:

Полнота. Если доказывающая сторона следует протоколу с правильным секретом, честная проверяющая убеждается в истинности утверждения. Иными словами, протокол всегда работает.

Корректность. Если доказывающая сторона не знает секрета, она не может убедить честную проверяющую в ложном утверждении, кроме как с пренебрежимой вероятностью. Иными словами,

доказывающие не могут мошенничать.

Нулевое разглашение. Проверяющая не узнает ничего, кроме того, что утверждение истинно. В частности, она не может узнать ничего о секрете доказывающей стороны.

Проверяющая "убеждается в истинности утверждения", потому что обе стороны согласны: протокол обладает этими тремя свойствами, и доказывающая не смогла бы завершить его для ложного утверждения, например не зная заявленного решения.

Модели атакующего

Обе стороны могут быть атакующими, пытающимися нарушить соответственно корректность и нулевое разглашение.

Злонамеренные доказывающие. Они хотят доказать ложное утверждение, например неверно убедить проверяющую сторону, что знают решение трудной задачи. Такой атакующий может отклоняться от протокола, пытаясь обмануть проверяющую сторону. На практике это главная угроза для ZKP в приложениях наподобие конфиденциального исполнения программ.

Злонамеренные проверяющие. Они хотят извлечь сведения о секрете (свидетеле), нарушив нулевое разглашение. Модели различают пассивных (честных, но любопытных) и активных, могущих произвольно отклоняться от протокола, проверяющих атакующих.

Злонамеренная проверяющая может оспаривать полноту, заявляя, что не убеждена в истинности утверждения. На практике это не проблема: доказывающая может повторить протокол для других, честных проверяющих, которые разоблачат лживую.

Протокол Шнорра

Клаус-Петер Шнорр, создавший одноименную схему подписи, также описал похожую конструкцию - доказательство с нулевым разглашением знания дискретного логарифма. Оно лежит в основе подписей Шнорра и EdDSA, а также многих более сложных ZKP.

Протокол Шнорра за три шага доказывает знание `a`, такого что `aG=A`, то есть дискретного логарифма `A` по основанию `G`:

1. **Фиксация.** Доказывающая выбирает случайное `r` и отправляет `R=rG`.
2. **Вызов.** Проверяющая отправляет случайное `c`.
3. **Ответ.** Доказывающая отправляет `s=r+ca`, а проверяющая принимает доказательство тогда и только тогда, когда `sG=R+cA`.

> **Примечание.** Такой трехшаговый протокол с фиксацией, вызовом и ответом называется сигма-протоколом из-за формы заглавной греческой буквы сигма (`Sigma`).

Полноту протокола проверить проще всего: если `aG=A`, то

```
sG=(r+ca)G=rG+caG=R+cA,
```

что и является условием проверки доказывающей стороны.

Для проверки корректности - того, что доказывающая должна знать `a`, - представьте, что она дважды применяет одно `r` в двух выполнениях протокола. Проверяющая получит два ответа `s_1=r+c_1a` и `s_2=r+c_2a` для двух разных вызовов `c_1` и `c_2`. Тогда

```
s_1-s_2=r+c_1a-r-c_2a=a(c_1-c_2),
```

a деление на c_1-c_2 даст a. Проверка sG=R+cA гарантирует, что s=r+ca, и потому при правильном выполнении доказывающая должна знать a. Такое логическое рассуждение называется экстрактором знания и служит главным приемом доказательства корректности ZKP.

Нулевое разглашение протокола также можно доказать с помощью симулятора - алгоритма, создающего сообщения (или транскрипт обмена), неотличимые от сообщений реального ZKP. В отличие от настоящей доказывающей стороны, симулятор не обязательно знает доказываемый секрет (свидетель), но его сообщения все равно выглядят допустимыми и убедительными для проверяющей.

В протоколе Шнорра симулятор работает в обратном направлении: сначала выбирает случайный ответ s, показывая, что настоящий s "столь же случаен", как чисто случайный. Затем он выбирает случайный вызов c и вычисляет исходную фиксацию R=sG-cA. Доказательство нулевого разглашения показывает, что эти три значения неотличимы от значений в реальном выполнении, хотя для их создания секрет a не требуется. В случае Шнорра нужно предположить, что проверяющая следует протоколу и выбирает случайное c.

Неинтерактивные доказательства

Протокол Шнорра интерактивен: доказывающая сторона отправляет первое сообщение, проверяющая отвечает вызовом, а затем доказывающая посылает ответ - стороны взаимодействуют в трех раундах. Но что, если проверяющая не может посылать сообщения и желает лишь получить одно, которое ее убедит? Как из интерактивного протокола создать неинтерактивное доказательство?

Еще раз посмотрим на протокол Шнорра, где проверяющая посылает случайный вызов c, который должен быть непредсказуем для доказывающей. Если она знает c до отправки R, она может смошенничать: выбрать произвольное s, вычислить R=sG-cA, отправить R, а затем ответить значением s. Проверка пройдет, хотя секрет a не использовался.

Как сделать c непредсказуемым без взаимодействия с проверяющей? Хитрость в том, чтобы получить c из R хеш-функцией. Псевдослучайное поведение хеш-функций не позволяет найти пару (R,c), удовлетворяющую sG=R+cA.

Для создания неинтерактивного доказательства с нулевым разглашением (NIZK) знания a по протоколу Шнорра доказывающая действует так:

1. **Фиксация.** Доказывающая выбирает случайное r и вычисляет R=rG.
2. **Вызов.** Доказывающая вычисляет c=H(G || R || A). Значения G и A нужно включить, чтобы связать генерацию c с параметром-генератором G и открытым ключом A доказывающей.
3. **Ответ.** Доказывающая вычисляет s=r+ca и создает доказательство как кодировку R и s.

Получив от доказывающей с открытым ключом A доказательство (R,s), проверяющая повторно вычисляет c=H(G || R || A) и проверяет sG=R+cA.

Прием замены создаваемых проверяющей вызовов хешированием данных протокола формализован преобразованием Фиата-Шамира - общим методом превращения интерактивного протокола в неинтерактивный. Для его применения случайные вызовы проверяющей должны быть независимы от сообщения доказывающей и быть открытыми (несекретными) значениями.

zkSNARK

Поговорим о виде ZKP, который благодаря своей мощности и эффективности широко применяется в блокчейнах. Например, zkSNARK лежат в основе платформы конфиденциальных транзакций Zcash: они доказывают, что некоторая сумма списана с одного счета и зачислена на другой, не раскрывая сумм и не требуя запретительно больших объемов вычислений или хранения.

zkSNARK - это вид неинтерактивного доказательства знания со свойством нулевого разглашения (zk), а SNARK расшифровывается так:

Краткое (succinct). Доказательство очень мало по сравнению с размером утверждения и секрета. Оно может быть сопоставимо с логарифмом размера утверждения или даже быть постоянного размера - всегда одинакового независимо от размера утверждения.

Неинтерактивное (noninteractive). SNARK - неинтерактивный аргумент знания, обычно полученный преобразованием Фиата-Шамира из интерактивного. Проверяющая не должна посылать сообщения доказывающей.

Аргумент (argument). Аргумент знания - вычислительно безопасное, то есть условно безопасное доказательство. Против атакующего с бесконечной вычислительной мощностью оно не было бы безопасным, что обычно считается приемлемым ограничением.

Знания (of knowledge). SNARK обладает полнотой и корректностью как аргумент знания.

Кроме того, создание и проверка zkSNARK должны быть вычислительно эффективными.

Кажется противоинтуитивным, что краткое доказательство может показать знание решения задачи, описание которой не помещается в его размер. Например, как доказать "я знаю решение уравнения `f(x)=0`", если `f(x)` произвольного размера?

С теоретической точки зрения краткие доказательства имеют смысл, потому что они должны сообщать лишь о знании некоторого решения и в общем о правильности утверждения, а не о решении и секретах: нулевое разглашение должно оставить проверяющей только эту информацию.

В 2016 году криптограф Йенс Грот опубликовал статью "On the Size of Pairing-Based Non-interactive Arguments" ("О размере неинтерактивных аргументов на основе паринговв", eprint.iacr.org), описавшую исключительно эффективный zkSNARK. Доказательство состояло всего из трех элементов группы и проверялось тремя парингами того же типа, что в BLS. Этот прорывной результат принял протокол Zcash, и он стал основой для нескольких других zkSNARK. Обычно его называют просто Groth16.

От утверждений к доказательствам

zkSNARK - одни из самых сложных криптографических конструкций. Один из самых сложных и дорогих шагов - арифметизация, преобразующая доказываемое утверждение в фиксированное число полиномиальных уравнений, обычно вида

```
a_0+a_1x+a_2x^2+a_3x^3+...+a_nx^n
```

для многочлена степени `n`, где `a_i` - элементы некоторого конечного кольца или поля. Затем алгоритм доказывающей обрабатывает многочлены и создает zkSNARK.

Арифметизация следует общему порядку:

1. Описать утверждение формальными обозначениями, например программой, уравнениями или логическими формулами.

2. Превратить формальное выражение в схему, которая получает выход из входов последовательным применением вентилей. Они похожи на логические вентили булевой или электронной схемы, но могут быть алгебраическими операциями наподобие сложения и умножения.
3. Превратить схему в структурированный список ограничений согласно системе ограничений zkSNARK. Ограничения представляют собой списки условий, которым должен удовлетворять вход, чтобы утверждение было истинным.

Утверждение может быть столь простым, как "я знаю целые x и y, удовлетворяющие `x^3+y^2+xy+55=0 mod (57)`". Такое формальное уравнение завершает шаг 1. Для шага 2 его разбивают на простые операции с двумя операндами, как требует Groth16, вводя промежуточные `v_0,v_1,...,v_6`:

```
Задать v0 = x × x.
Задать v1 = x × v0; таким образом, v1 = x³.
Задать v2 = y × y.

Задать v3 = x × y.
Задать v4 = v1 + v2.
Задать v5 = v4 + v3.
Задать v6 = v5 + 55; таким образом, v6 = x³ + y² + xy + 55.
```

Такой перевод длинного уравнения в ряд малых называется уплощением.

Затем доказывающая преобразует эти операции - схему - в набор математических структур, по которым строятся многочлены. Длинные многочлены в конце сжимаются в доказательство с помощью случайности - в частности, понятия вероятностно проверяемого доказательства (PCP), важного открытия теории сложности. Оно убеждает проверяющую, что многие ограничения выполнены, фактически проверяя лишь несколько и сохраняя нулевое разглашение.

Подробнее о тонкостях арифметизации можно узнать, исследовав два главных подхода zkSNARK: системы ограничений ранга 1 (R1CS) и алгебраические промежуточные представления (AIR).

Наконец, следует различать неуниверсальные и универсальные системы zkSNARK. В первых доказывающая работает с конкретным, заранее заданным утверждением, а фаза настройки создает параметры только для него. Универсальные системы, например Marlin и Plonk, принимают утверждение на вход и создают для него доказательство. Они гибче, но сложнее в построении и требуют больше вычислений.

Что может пойти не так

zkSNARK могут страдать от тех же классов проблем, что и пороговые подписи: от небезопасного протокола до дефектов реализации. Ошибки возникают на разных этапах - от определения утверждения до арифметизации и вычисления доказательства. Они могут нарушить полноту, корректность или нулевое разглашение. Чаще всего главный риск касается корректности - возможности обмануть проверяющую. Это объясняется и возможным ущербом, и тонкостью ошибок, тогда как утечка "знания" менее вероятна, особенно если доказательство должно остаться допустимым.

В следующих примерах мы сосредоточимся на протоколе Шнорра. Проблемы сравнительно просты, но в более сложных системах они могут быть гораздо тоньше.

Напомню: неинтерактивный протокол Шнорра доказывает знание a проверяющей, знающей `A=aG`, посылая `s=r+ca` и `R=rG` для случайного `r` и `c=H(G || R || A)`. После повторного вычисления `c` проверяющая проверяет `sG=R+cA`. В интерактивной версии `c` выбирает случайно проверяющая.

Недостаточное хеширование Фиата-Шамира

Представьте, что вместо `c=H(G || R || A)` неинтерактивное доказательство Шнорра использует `c=H(G || A)`, делая `c` независимым от `R`. Атакующий выбирает произвольное `s` и вычисляет `R=sG-cA`. Получившееся доказательство из `s` и `R` допустимо, хотя атакующий не знал `a`: корректность протокола нарушена.

Равным образом, если `c=H(G || R)` и из хешируемых данных исключен `A`, атака возможна: атакующий выбирает произвольные `R` и `s` и вычисляет `B=(1/c)*(sG-R)`, удовлетворяющую `sG=R+cB`. Так он получает доказательство знания дискретного логарифма `B` - то есть `b`, такого что `B=bG`, - хотя не знает его.

Эти атаки показывают, как важно включать все необходимые значения во вход хеш-функции при преобразовании Фиата-Шамира.

Атаки повторного воспроизведения

Повторное воспроизведение - тривиальная, но потенциально опустошительная атака. Узнав неинтерактивное доказательство знания, атакующий может послать его другой стороне и заявить, что создал его, присвоив себе заслугу за доказанное знание.

Этого можно избежать, связав доказательство с личностью доказывающей и включив ее открытый ключ в хешируемые данные. Чтобы та же сторона не могла повторно применить его позднее, доказательство можно связать с идентификатором сеанса или меткой времени, включив их в данные для хешей Фиата-Шамира.

Повторное использование случайности

Рассмотрим интерактивный протокол Шнорра, где проверяющая выбирает случайное `c`. Если дефектный генератор псевдослучайных чисел доказывающей дважды повторит одно одноразовое значение `r`, наблюдающий обмен атакующий по двум доказательствам `s_1=r+c_1a` и `s_2=r+c_2a` для `c_1!= c_2` восстановит секрет:

```
a=(s_1-s_2)/(c_1-c_2).
```

По-настоящему серьезная криптография

В этой заключительной главе мы рассмотрели одни из самых увлекательных на момент написания тем криптографии и с теоретической, и с практической точки зрения. Однако мы лишь коснулись поверхности, особенно в области ZKP - активной области исследований и инженерной работы с широкими приложениями за пределами блокчейнов. Но крутая криптография - не панацея для блокчейнов. От протоколов многосторонних вычислений, таких как пересечение конфиденциальных множеств (PSI), до гомоморфного шифрования для конфиденциальной оценки моделей ИИ, новые приложения и сценарии требуют лучших и более быстрых криптографических функций.

Мы наблюдаем золотой век криптографии с беспрецедентным сближением теоретических принципов и практических приложений. Эта синергия дает почти волшебные решения для некоторых из самых сложных проблем безопасности и конфиденциальности. Тем не менее еще нужно решить существенные задачи, особенно в правовой и регуляторной сферах. Крайне важно, чтобы технологи и политики тесно сотрудничали для преодоления этих трудностей и пытались понять позиции друг друга. Я надеюсь, что эта книга, и особенно ее последняя глава, поможет развеять мифы о криптографии, сделав ее доступнее и менее загадочной для всех читателей.