Оглавление

Предисловие ... 6

Введение ... 7

Глава 1. Алгоритмы хеширования ... 9

1.1 Основные понятия и определения ... 10
1.1.1 Структура алгоритмов хеширования ... 10
1.1.2 Надстройки над алгоритмами хеширования ... 14
1.2 Методы криптоанализа и атаки на алгоритмы хеширования ... 18
1.2.1 Цели атак на алгоритмы хеширования ... 19
1.2.2 Атаки методом «грубой силы» ... 21
1.2.3 Словарные атаки и цепочки хеш-кодов ... 22
1.2.4 Радужные таблицы ... 26
1.2.5 Парадокс «дней рождения» и поиск коллизий ... 27
1.2.6 Дифференциальный криптоанализ ... 30
1.2.7 Алгебраический криптоанализ ... 34
1.2.8 Атаки, использующие утечки данных по побочным каналам ... 35
1.2.9 Другие виды атак ... 35
1.3 Наиболее известные алгоритмы хеширования ... 38
1.3.1 Алгоритмы семейства MD ... 38
1.3.2 Алгоритмы семейства RIPEMD ... 59
1.3.3 Алгоритмы семейства SHA ... 69
1.3.4 Отечественные стандарты хеширования ... 84

Глава 2. Алгоритмы электронной подписи на эллиптических кривых ... 91

2.1 Математические основы ... 91
2.2 Эллиптические кривые ... 95
2.2.1 Определение эллиптической кривой ... 95
2.2.2 Основные операции над точками эллиптической кривой ... 96
2.2.3 Основные характеристики эллиптической кривой ... 99
2.2.4 Примеры эллиптических кривых ... 101
2.2.5 Задача дискретного логарифмирования в группе точек эллиптической кривой ... 105
2.2.6 Альтернативные формы представления эллиптических кривых ... 107
2.3 Основные алгоритмы электронной подписи ... 111
2.3.1 Алгоритм ECDSA ... 111
2.3.2 ГОСТ Р 34.10–2012 ... 112

2.3.3 Некоторые особенности алгоритмов ECDSA и ГОСТ Р 34.10–2012 114
2.3.4 Алгоритм EdDSA 116
2.3.5 Алгоритм BLS 118

Глава 3. Основные принципы работы блокчейн-технологий 122

3.1 Базовые механизмы блокчейн-систем 123
3.1.1 Транзакции 123
3.1.2 Упаковка транзакций в блоки 127
3.1.3 Применение деревьев Меркля при формировании блоков 130
3.2 Механизмы консенсуса 131
3.2.1 Консенсус доказательства работы Proof of Work 131
3.2.2 Консенсус доказательства владения долей Proof of Stake 135
3.2.3 Консенсус на основе решения задачи византийских генералов 136
3.2.4 Другие механизмы достижения консенсуса 137
3.3 Выстраивание цепочки блоков 139
3.3.1 Принципы формирования цепочки 139
3.3.2 Ветвления цепочки блоков 142
3.4 Смарт-контракт 145
3.5 Основные виды блокчейн-систем 148
3.5.1 Публичный блокчейн 148
3.5.2 Приватный блокчейн 149
3.6 Криптовалютные кошельки 150
3.6.1 Программы-кошельки 150
3.6.2 Аппаратные кошельки 152

Глава 4. Основные блокчейн-платформы 153

4.1 Биткойн 153
4.1.1 Введение в устройство блокчейн-системы Биткойн 154
4.1.2 Особенности механизма консенсуса в системе Биткойн 156
4.1.3 Форки в системе Биткойн 156
4.1.4 Транзакции 159
4.1.5 Кошельки в системе Биткойн 203
4.1.6 Создание и использование иерархических детерминированных ключей 206
4.2 Эфириум 209
4.2.1 Глобальное состояние 209
4.2.2 Консенсус 210
4.2.3 Газ 211
4.2.4 Адреса и кошельки 212
4.2.5 Транзакции 213
4.2.6 Структура блока 213
4.2.7 Эволюция системы Эфириум 214
4.2.8 Основная и тестовые сети платформы Эфириум 218
4.2.9 Запуск сети Эфириум 219
4.2.10 Смарт-контракты в системе Эфириум 234
4.3 Hyperledger 245

4.3.1 Основные особенности системы 245
4.3.2 Проекты экосистемы Hyperledger 246
4.3.3 Архитектура Hyperledger Fabric 247
4.3.4 Пример смарт-контракта для Hyperledger 248
4.4 Обзор других платформ 253
4.4.1 EOSIO 253
4.4.2 Краткий обзор прочих блокчейн-платформ 255
4.4.3 Обзор отечественных решений 257

Приложение 1. Таблицы констант алгоритмов хеширования 261

П1.1 Таблица замен алгоритма MD2 261
П1.2 Индексы используемых в итерациях слов блока сообщения алгоритма MD4 262
П1.3 Константы алгоритма MD5 263
П1.4 Константы алгоритма MD6 267
П1.5 Константы алгоритмов семейства SHA-2 268
П1.6 Раундовые константы алгоритмов семейства SHA-3 270
П1.7 Константы алгоритма ГОСТ Р 34.11–2012 271

Список сокращений 275

Перечень рисунков 281

Перечень таблиц 286

Перечень источников 289

Предисловие

В последние десятилетия криптографические методы проникают в самые различные сферы нашей жизнедеятельности, связанные с передачей, обработкой и хранением информации. Цепная запись данных, распределенные реестры, интернет вещей, облачные вычисления в той или иной мере используют криптографические алгоритмы и протоколы. Появление криптовалюты биткойн привлекло внимание научного сообщества к технологии блокчейн, или технологии цепной записи данных. Последовала разработка блокчейн-платформ для использования в самых различных областях: финансовой и банковской сфере, медицине, торговле и т. п. При этом успешное применение технологий, основанных на криптографии, невозможно без понимания математических основ, на которых базируются криптографические алгоритмы, используемые в данной технологии.

Приведенная в этой книге информация позволяет получить представление по наиболее важным вопросам построения блокчейн-платформ, таким как принципы построения и использования функций хеширования, схем электронной подписи на основе эллиптических кривых, механизмов консенсуса. Приведенные математические основы используемых в технологии блокчейн криптографических алгоритмов помогают глубже понять заложенные в ней механизмы обеспечения безопасности информации. Данная книга может быть интересна самому широкому кругу читателей, желающих понять принципы построения и использования блокчейн-технологий.

Сергей Васильевич Матвеев,
эксперт ТК-26 «Криптографическая защита информации»

Введение

С момента появления блокчейн-технологий прошло менее 15 лет, но их активное развитие в течение этого времени предопределило вхождение данных технологий в весьма различные сферы деятельности.

Основной сферой применения блокчейн-технологий можно считать финансовую: они лежат в основе криптовалют, которые, похоже, уже достаточно прочно вошли в нашу жизнь. Буквально в последний год мы могли наблюдать всплеск интереса к криптовалютам после многократного удорожания основной из них – биткойна – в течение 2020 – начала 2021 года.

При этом постоянно модернизируются и совершенствуются как сама платформа Биткойн и лежащая в ее основе блокчейн-система, так и данные технологии в принципе. Их развитие способствует и развитию множества смежных технологий и направлений: от криптографических алгоритмов до вычислительных ресурсов, применяемых пользователями подобных систем.

Помимо финансового сектора, блокчейн-технологии востребованы в различных системах государственных организаций, в частности:

- ведение различных реестров, например государственной регистрации прав на недвижимое имущество, землю и т. п.;
- выпуск цифровых удостоверений личности на основе блокчейн-технологий;
- удаленное голосование, опробованное, в частности, в России в 2019 и 2020 годах.

Можно предполагать, что в дальнейшем приведенный выше, далеко не полный перечень применений блокчейнов будет только расширяться.

Отметим, что распределенные реестры были известны и ранее, но именно блокчейн-технологии, обеспечивающие, с одной стороны, возможность модификации общих данных различными пользователями распределенных систем и, с другой стороны, контроль целостности и непротиворечивости данных на основе определенных правил, предопределили масштабное развитие подобных технологий и появление таких принципиально новых направлений, как криптовалюты.

Про блокчейн-технологии, особенно в части их применений в криптовалютах, издано достаточно много книг. Специфика этой книги в том, что при ее создании мы изначально ставили своей целью рассмотреть и проанализировать именно криптографические механизмы, лежащие в основе блокчейн-технологий и предопределяющие их основные качества, способствующие столь бурному развитию и предполагаемому в будущем широчайшему применению данных технологий.

Понимая и разделяя интерес многих потенциальных читателей к криптовалютам, мы не обошли их стороной, но рассмотрели именно с точки зрения реализованных в них криптоалгоритмов и прочих методов, обеспечивающих технические составляющие безопасного использования криптовалют.

В первой главе книги описаны алгоритмы хеширования, обеспечивающие контроль целостности данных в блокчейне. Рассмотрены основные принципы данных алгоритмов, возможные проблемы при их реализации и использовании, включая известные атаки на алгоритмы хеширования. Приведено подробное описание наиболее известных алгоритмов хеширования, включая используемые в распространенных блокчейн-платформах.

Вторая глава также посвящена криптографическим алгоритмам – на этот раз алгоритмам электронной подписи, являющимся одним из важнейших элементов, обеспечивающих связь в цепочках данных блокчейна, и не только. Рассмотрены эллиптические кривые, лежащие в основе современных алгоритмов электронной подписи и наиболее часто применяемые из данных алгоритмов.

Третья глава описывает базовые механизмы построения цепочек данных – основы блокчейн-технологий. Значительная часть главы посвящена описанию различных механизмов достижения консенсуса, легитимизирующих действия пользователей с данными блокчейна.

Наконец, в последней главе рассмотрены примеры построения блокчейн-платформ на основе алгоритмов и методов, описанных в предыдущих главах. В частности, дано подробное описание системы Биткойн, обеспечивающей оборот одноименной криптовалюты, наиболее широко используемой в мире.

Надеемся, что изложенная в нашей книге информация оправдает ваши ожидания от книги, окажется интересной и принесет пользу в вашей деятельности.

Авторы выражают глубокую признательность известному специалисту по прикладной криптографии Олегу Геннадьевичу Тараскину (компания Waves) за предоставленную для публикации в данной книге главу 2, без материала которой книга была бы неполной, а также за множество полезных замечаний, позволивших значительно улучшить книгу.

Авторы также благодарны эксперту технического комитета по стандартизации «Криптографическая защита информации» (ТК-26) Сергею Васильевичу Матвееву за предисловие к книге и ценные замечания по ее материалу.

Будем рады вашим письмам по изложенным в книге вопросам, а также замечаниям к содержимому книги и предложениям по ее возможному усовершенствованию. Адреса электронной почты для связи с авторами:

serg@panasenko.ru (Сергей Панасенко) и
jekky82@mail.ru (Евгения Ищукова).

Глава 1

Алгоритмы хеширования

Алгоритмы хеширования (или функции хеширования, хеш-функции) позволяют по определенным правилам выполнить свертку входных данных произвольной длины в битовую строку фиксированного размера, называемую хеш-кодом [15] (распространены также термины «хеш» или «хеш-значение»).

Фактически хеш-функции выполняют контрольное суммирование данных, которое может происходить как с участием некоего секретного ключа, так и без него. Обычно алгоритмы ключевого хеширования представляют собой надстройки над алгоритмами хеширования, не использующими ключ. Однако существуют и такие хеш-функции, которые изначально разрабатывались с учетом использования секретного ключа в качестве дополнительного параметра преобразований.

Такое контрольное суммирование достаточно широко применяется в области защиты компьютерной информации, в том числе:

- для подтверждения целостности данных;
- для свертки данных перед вычислением или проверкой их электронной подписи;
- в различных протоколах аутентификации пользователей;
- в процедурах генерации псевдослучайных последовательностей и производных ключей.

Алгоритмы хеширования применяются и с другими целями, не относящимися к задачам защиты информации. В частности, они применяются для вычисления уникальных идентификаторов данных и построения на их основе хеш-таблиц, существенно ускоряющих поиск требуемых данных в больших массивах. Однако в подобных случаях к алгоритмам хеширования предъявляются иные требования, чем при их криптографических применениях.

Алгоритмы хеширования активно используются в блокчейн-технологиях, наиболее часто – для свертки данных перед их подписанием электронной подписью. В ряде случаев хеш-функции используются и с другими целями – например, в биткойне для подтверждения проделанной работы используются найденные (получаемые путем перебора) хеш-коды специального формата – с количеством лидирующих битовых нулей не менее заданного.

Конкретные применения алгоритмов хеширования в технологиях блокчейна будут описаны в главах 3 и 4, а в данной главе рассмотрим подробно структуру алгоритмов хеширования, предъявляемые к ним требования, а также основные методы и результаты их криптоанализа.

1.1 Основные понятия и определения

Функции хеширования позволяют выполнить однонаправленное преобразование входного массива произвольного размера в выходную битовую строку (хеш-код) фиксированного размера.

Криптографические хеш-функции должны обладать, как минимум, следующими свойствами [44, 202].

1. Хеш-код сообщения должен однозначно соответствовать сообщению и должен изменяться при любой модификации сообщения.
2. Должно быть вычислительно сложно найти прообраз, т. е. такое сообщение M, хеш-код которого был бы равен заданному значению h:

   $$h = \mathrm{f}(M),$$

   где f() – функция хеширования.
3. Должно быть вычислительно сложно найти второй прообраз, т. е. такое сообщение M_2, хеш-код которого был бы равен хеш-коду заданного сообщения M_1:

   $$\mathrm{f}(M_1) = \mathrm{f}(M_2).$$

4. Должно быть вычислительно сложно найти коллизию, т. е. такие два сообщения M_1 и M_2, хеш-коды которых были бы эквивалентны.

1.1.1 Структура алгоритмов хеширования

Хотя все множество алгоритмов хеширования достаточно разнообразно по структурам их формирования, наиболее известные алгоритмы хеширования основаны на нескольких типовых структурах, которые рассмотрим в данном разделе далее.

Схема Меркля–Дамгорда

Многие из широко используемых алгоритмов хеширования имеют схожую между собой структуру, которая была предложена в 1988–1989 гг. независимо двумя известными криптологами: Ральфом Мерклем (Ralph Merkle) [177] и Айвеном Дамгордом (Ivan Damgård) [102]. Она получила название «схема Меркля–Дамгорда» (Merkle-Damgård construction) – см. рис. 1.1.

В соответствии со схемой Меркля–Дамгорда работают, в частности, алгоритмы MD4 [210], MD5 [212] и SHA [125], а также отечественный стандарт хеширования ГОСТ Р 34.11–2012 [15], которые будут подробно рассмотрены в этой главе далее.

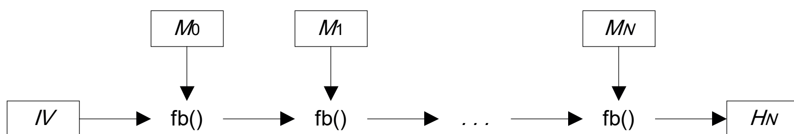

Рисунок 1.1. Схема Меркля–Дамгорда

Хешируемое сообщение M разбивается на блоки определенной длины (например, по 512 байт в алгоритме SHA-1) $M_0 \ldots M_N$. Выполняется (по-разному

в различных алгоритмах) дополнение сообщения *M* до размера, кратного данной длине блока (при этом количество блоков может увеличиться – см., например, описание алгоритма SHA далее).

Каждый *i*-й блок сообщения обрабатывается функцией сжатия fb() (функция сжатия является криптографическим ядром алгоритма хеширования), причем данная функция накладывает результат этой обработки на текущий (промежуточный) хеш-код H_{i-1} (т. е. результат обработки предыдущих блоков сообщения):

$$H_i = \mathrm{fb}(H_{i-1}, M_i).$$

Результатом работы алгоритма хеширования hash() (т. е. хеш-кодом сообщения *M*) является значение H_N:

$$\mathrm{hash}(M) = H_N.$$

Начальное значение H_{-1} обычно является константным и различным в разных алгоритмах хеширования; оно часто обозначается как *IV* (от Initialization Vector – вектор инициализации).

Алгоритмы хеширования на основе криптографической губки

Схема алгоритмов хеширования на основе «криптографической губки» (cryptographic sponge) была предложена рядом криптологов в работе [69]. Авторами данной схемы являются Гвидо Бертони (Guido Bertoni), Джоан Деймен (Joan Daemen), Михаэль Петерс (Michaël Peeters) и Жиль Ван Аске (Gilles Van Assche).

Алгоритмы хеширования, использующие конструкцию криптографической губки, содержат две фазы преобразований (рис. 1.2):

- «впитывание» (absorbing) – поблочная обработка входного сообщения, в процессе которой внутреннее состояние алгоритма хеширования «впитывает» свертку данных очередного блока;
- «выжимание» (squeezing) – сжатие внутреннего состояния для извлечения из него результирующего хеш-кода.

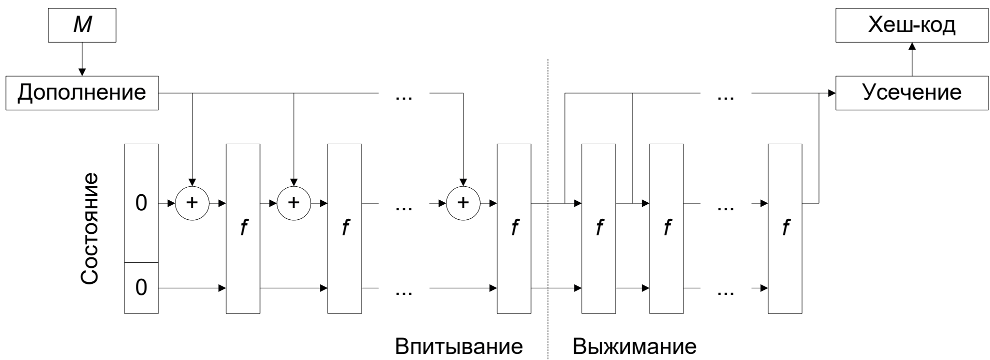

Рисунок 1.2. Фазы преобразований криптографической губки

Конструкция криптографической губки, по мнению ее авторов, имеет как минимум следующие преимущества [69, 71]:

- возможность точного и доказуемого определения минимальной трудоемкости основных атак на криптоалгоритмы, основанные на данной конструкции;
- относительная простота криптоалгоритмов;
- легкое построение алгоритмов с переменным размером входных и выходных данных;
- универсальность конструкции – с ее помощью можно создавать криптоалгоритмы различного назначения;
- легкое увеличение расчетного уровня криптостойкости алгоритма за счет изменения его параметров и снижения его быстродействия.

Надстройки над алгоритмами блочного шифрования

Существуют также варианты создания стойких однонаправленных хеш-функций на основе алгоритмов блочного шифрования. В частности, в работе [202] подробно описаны 12 алгоритмов, представляющих собой хеширующие надстройки над блочными шифрами.

Рассмотрим две из таких надстроек более подробно. Первая из них – это алгоритм Матиаса–Мейера–Осиса (Matyas-Meyer-Oseas – MMO), функция сжатия которого представлена на рис. 1.3. В данном алгоритме нижележащий блочный шифр используется следующим образом [174]:

- блок хешируемых данных M_i подается на вход алгоритма шифрования в качестве открытого текста (*P*);
- ключом блочного шифра (*K*) служит текущее состояние алгоритма хеширования H_{i-1}, обработанное некоторой функцией g() (см. далее);
- на результат шифрования (*C*) блока M_i на ключе g(H_{i-1}) накладывается операцией XOR сам блок M_i, в результате чего получается новое состояние алгоритма хеширования H_i.

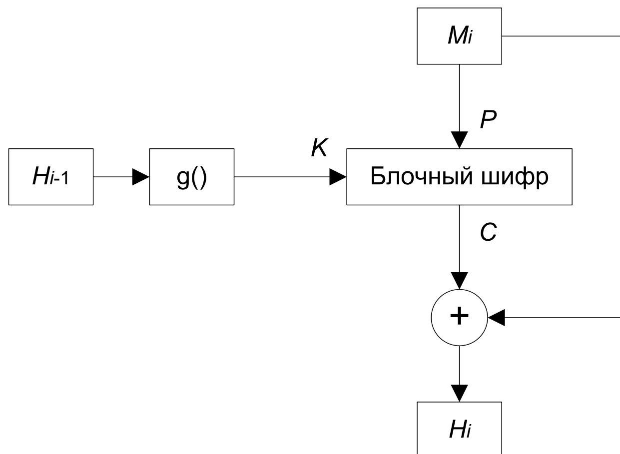

Рисунок 1.3. Схема алгоритма MMO

Поскольку размеры блока шифруемых данных и ключа алгоритма шифрования могут быть различными (и чаще всего они действительно различаются),

а размер состояния алгоритма хеширования является фиксированным, в алгоритме MMO существует необходимость в функции g(), задача которой состоит лишь в адаптации значения H_{i-1} под требуемый размер ключа шифрования. Так как функция g() не влияет на безопасность алгоритма, она может быть максимально простой и, например, выполнять только дополнение H_{i-1} до требуемого размера [246].

Формально алгоритм MMO представляется так:

$$H_i = \mathrm{E}(M_i, g(H_{i-1})) \oplus M_i,$$

где E(P, K) – блочный шифр, шифрующий открытый текст P на ключе K.

Существует также вариант алгоритма MMO, который вместо операции XOR использует сложение 32-битовых субблоков M_i и C по модулю 2^{32}. В работе [217] такой вариант называется предпочтительным, в частности для алгоритмов MD4 и MD5.

В качестве блочного шифра может использоваться также функция сжатия какого-либо алгоритма хеширования, поскольку, как и блочный шифр, функции сжатия алгоритмов хеширования обычно принимают на вход два параметра (состояние и хешируемый блок данных вместо ключа шифрования и шифруемого блока), выдавая в качестве результата модифицированное состояние. В этом случае описанные в данном разделе схемы можно рассматривать как надстройки, усиливающие криптографические свойства нижележащего алгоритма хеширования.

Еще один вариант формирования алгоритма хеширования на основе блочного шифра – алгоритм Миягучи–Пренеля (Miyaguchi-Preneel), который отличается от предыдущего только тем, что предыдущее состояние алгоритма хеширования также участвует в операции XOR с блоком сообщения и результатом его зашифрования (рис. 1.4) [174]:

$$H_i = \mathrm{E}(M_i, g(H_{i-1})) \oplus M_i \oplus H_{i-1}.$$

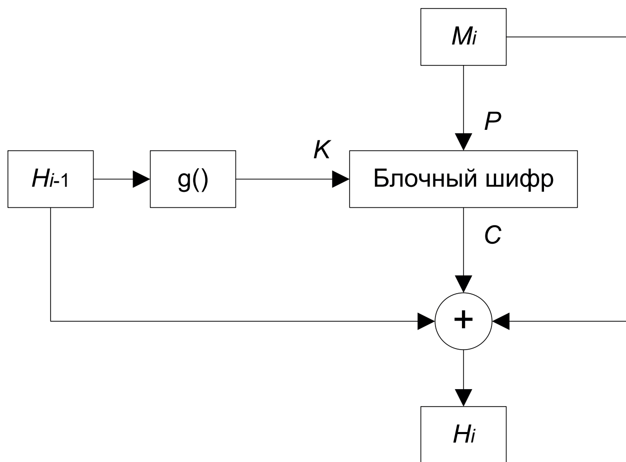

Рисунок 1.4. Схема алгоритма Миягучи–Пренеля

Аналогично предыдущей схеме, здесь также возможна (и рекомендуется для тех же алгоритмов MD4 и MD5 [217]) замена операции XOR на операцию сложения субблоков операндов по модулю 2^{32}.

1.1.2 Надстройки над алгоритмами хеширования

В свою очередь, алгоритмы хеширования достаточно часто используются в качестве основы для различных кодов аутентификации сообщений (Message Authentication Codes – MAC).

Коды аутентификации сообщений на основе алгоритмов хеширования и их варианты

Код аутентификации сообщения фактически представляет собой криптографическую контрольную сумму сообщения. MAC вычисляется на основе ключа и текста сообщения произвольной длины и используется для проверки целостности сообщения [246]. Для вычисления MAC используются ключевые алгоритмы хеширования (или ключевые надстройки над алгоритмами хеширования), алгоритмы блочного шифрования или специальные алгоритмы вычисления кодов аутентификации сообщений.

В отличие от алгоритмов электронной подписи (ЭП), в которых для вычисления ЭП используется закрытый ключ, а для проверки ЭП – вычисляемый из закрытого открытый ключ, при вычислении и проверке MAC применяется один и тот же секретный ключ. Следовательно, в отличие от ЭП, MAC не используется для установления авторства сообщения, поскольку секретный ключ принадлежит более, чем одному пользователю.

Схема использования MAC приведена на рис. 1.5.

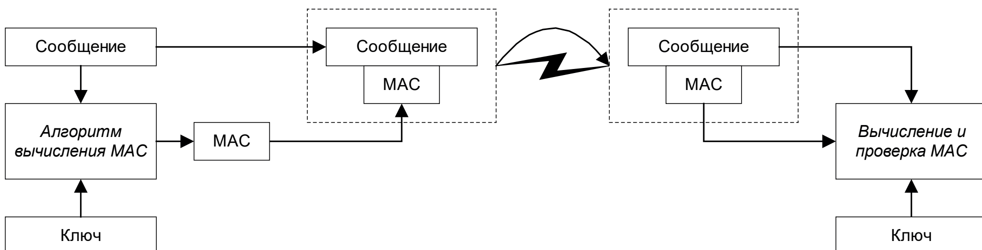

Рисунок 1.5. Схема использования MAC

Простейшие варианты создания MAC на основе произвольного бесключевого алгоритма хеширования выглядят так (при этом все три описанных далее варианта были признаны небезопасными) [202, 203]:

1. «Секретный префикс» (secret prefix):

$$MAC(k, m) = hash(k \,||\, m),$$

где:

- k – ключ;
- m – сообщение;
- hash() – нижележащая бесключевая хеш-функция.

2. «Секретный суффикс» (secret suffix):

$$MAC(k, m) = hash(m \| k).$$

3. «Конверт» (envelope):

$$MAC(k_1, k_2, m) = hash(k_1 \| m \| k_2),$$

где подключи k_1 и k_2 являются различными.

Более безопасным способом создания MAC на основе алгоритмов хеширования является алгоритм HMAC (Hash-based Message Authentication Code – код аутентификации сообщения на основе хеширования), который позволяет вычислять хеш-коды с использованием некоего секретного ключа с помощью практически произвольного бесключевого алгоритма хеширования [163].

В основе алгоритма HMAC лежит функция хеширования, которая позволяет вычислить код аутентификации сообщения следующим образом (см. рис. 1.6):

1. Размер ключа выравнивается с размером блока используемого алгоритма хеширования:
   - если ключ *key* длиннее блока, он укорачивается путем применения к нему используемого алгоритма хеширования hash (этот вариант не показан на рис. 1.6):

     $$key = hash(key);$$

     отметим, что размер выходного значения алгоритма хеширования обычно много меньше размера блока хешируемых данных – например, соответственно, 128 и 512 бит для алгоритма MD4; в [163] рекомендуется минимальный размер ключа, равный размеру выходного значения алгоритма хеширования;
   - если размер ключа меньше размера блока, то выровненный ключ *k* получается путем дополнения до размера блока нулевыми битами исходного (или укороченного) ключа *key*.
2. Выровненный ключ *k* складывается по модулю 2 с константой *C*1, которая представляет собой блок данных, заполненный байтами с шестнадцатеричным значением 36; аналогичным образом ключ *k* также складывается с константой *C*2, которая представляет собой блок данных, заполненный байтами с шестнадцатеричным значением 5C:

   $$ki = k \oplus C1;$$

   $$ko = k \oplus C2.$$

3. Вычисляется хеш-код *t* от результата конкатенации модифицированного ключа *ki* и сообщения *m*:

   $$t = hash(ki \| m).$$

4. Выходным значением алгоритма HMAC является хеш-код от результата конкатенации модифицированного ключа *ko* и полученного на предыдущем шаге значения *t*:

   $$hmac = hash(ko \| t).$$

Таким образом, при вычислении HMAC используемый алгоритм хеширования применяется дважды; каждый раз с участием модифицированного ключа. Размер выходного значения алгоритма HMAC равен размеру выходного значения алгоритма хеширования, а общая формула вычисления HMAC (без учета выравнивания ключа) выглядит следующим образом:

$$\mathrm{HMAC}(k, m) = \mathrm{hash}((k \oplus C2) \,||\, \mathrm{hash}((k \oplus C1) \,||\, m)).$$

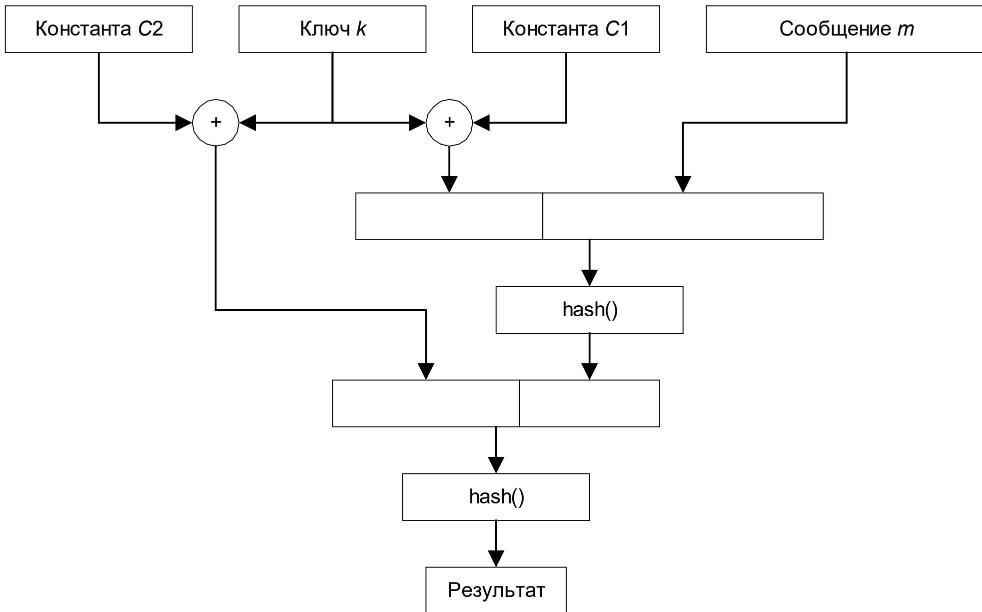

Рисунок 1.6. Вычисление HMAC

Алгоритмы, построенные с помощью HMAC, традиционно обозначают следующим образом:

- «HMAC-x», где «x» – используемый алгоритм хеширования, например HMAC-MD4;
- «HMAC-x-k» в тех случаях, где выходное значение алгоритма хеширования может быть усечено (т. е. может использоваться частично); в данном случае «k» – размер выходного значения алгоритма HMAC в битах; пример – алгоритм HMAC-SHA1-80 [163].

Стоит отметить, что быстродействие HMAC ненамного хуже, чем у используемой функции хеширования, что особенно заметно при хешировании длинных сообщений, поскольку само сообщение при использовании HMAC обрабатывается однократно [158].

Алгоритм NMAC (Nested Message Authentication Code – «вложенный» код аутентификации сообщения) был предложен авторами алгоритма HMAC в работе [62]. Данный алгоритм позволяет использовать при аутентификации сообщений два ключа: $k1$ и $k2$. Формула вычисления NMAC крайне проста:

$$\mathrm{NMAC}(k1, k2, m) = \mathrm{hash}(k2, \mathrm{hash}(k1, m)),$$

где второй параметр функции hash() обозначает хешируемое сообщение, а первый параметр – нестандартный вектор инициализации, в качестве которого используется ключ (соответственно, $k1$ и $k2$ обозначают уже выровненные до размера вектора инициализации ключи).

NMAC фактически является обобщением HMAC, поскольку ключи $k1$ и $k2$ можно использовать независимо, а можно представить как производные от единственного ключа k, вычисляемые следующим образом [158]:

$$k1 = \mathrm{h}(k \oplus C1);$$

$$k2 = \mathrm{h}(k \oplus C2),$$

где h() – функция сжатия используемого алгоритма хеширования; в данном случае применяется стандартный вектор инициализации алгоритма.

Схема NMAC приведена на рис. 1.7.

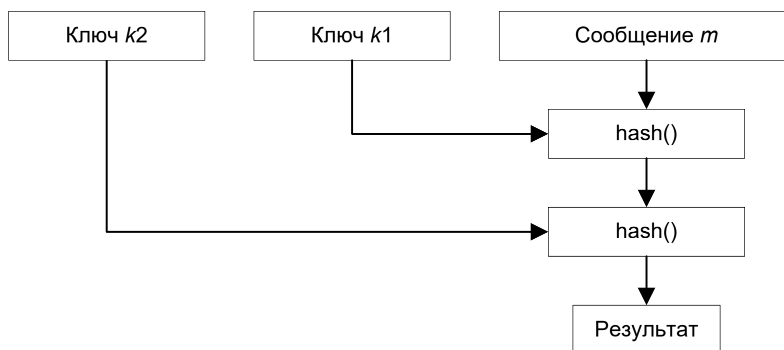

Рисунок 1.7. Вычисление NMAC

Хеширование с использованием деревьев Меркля

Дерево Меркля (Merkle tree) представляет собой бинарное дерево, листья которого содержат хеш-коды блоков данных, а узлы содержат хеш-коды от конкатенации хеш-кодов дочерних листьев или узлов.

Данная структура была предложена Ральфом Merклем в 1979 г. и запатентована в 1982 г. в патенте [178]. Общий вид дерева Меркля приведен на рис. 1.8.

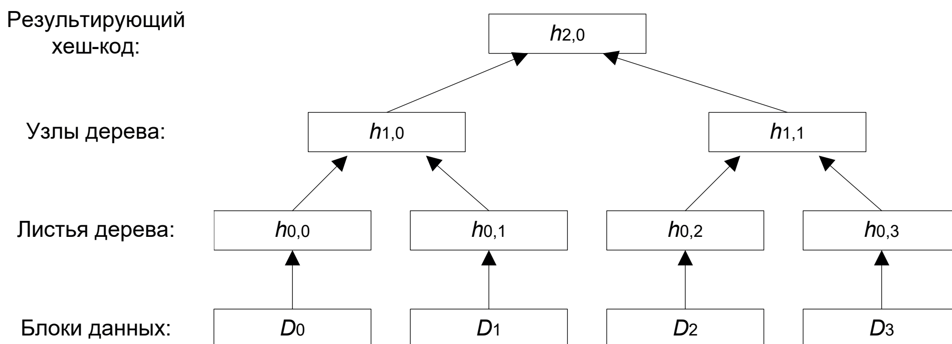

Рисунок 1.8. Дерево Меркля

В таком дереве вычисление хеш-кодов должно выполняться «снизу вверх» следующим образом:

1. Вычисление хеш-кодов данных для каждого листа дерева:

$$h_{0,i} = \text{hash}(D_i),$$

где:

- D_i – i-й блок данных;
- $h_{0,i}$ – хеш-код i-го блока данных;
- hash – используемая функция хеширования.

2. Поочередное вычисление хеш-кодов узлов дерева, начиная с более нижних уровней до вычисления корневого хеш-кода, который является результирующим:

$$h_{j,k} = \text{hash}(h_{j-1,2k} \,||\, h_{j-1,2k+1}),$$

где:

- $h_{a,b}$ – b-й хеш-код a-го уровня дерева;
- || – операция конкатенации.

Дерево Меркля активно применяется при хешировании данных в различных приложениях (включая ряд известных криптовалют – см. главу 4) в связи с наличием у него ряда положительных свойств (по сравнению с хешированием всех данных за один проход, т. е. одним вызовом функции hash), включая следующие:

- вычисление хеш-кодов с использованием дерева Меркля достаточно хорошо распараллеливается;
- при изменении одного из блоков данных не требуется пересчет хеш-кодов всех данных – достаточно пересчитать только хеш-коды, соответствующие той ветви дерева, которой принадлежит измененный блок данных;
- для проверки целостности одного из блоков данных также не требуется проверять целостность всех данных – достаточно проверить только соответствующую блоку ветвь дерева.

При большом количестве блоков данных пересчет одной ветви дерева по сравнению со всеми данными блоков требует значительно меньшей трудоемкости, в частности в криптовалютных применениях различия в трудоемкости могут достигать нескольких порядков.

Схожий с деревом Меркля структурный элемент применяется и непосредственно в ряде алгоритмов хеширования. В качестве примера таких алгоритмов можно привести алгоритм хеширования MD6, который будет подробно рассмотрен в данной главе далее.

1.2 Методы криптоанализа и атаки на алгоритмы хеширования

Рассмотрим в данном разделе цели, которые преследуют атакующие при анализе алгоритмов хеширования, а также существующие методы криптоанализа и некоторые из основных результатов атак на хеш-функции.

1.2.1 Цели атак на алгоритмы хеширования

Опишем основные и промежуточные цели атак на алгоритмы хеширования.

Основные цели атак на алгоритмы хеширования

Нахождение коллизии

Коллизией (collision) является ситуация, в которой два различных сообщения $m1$ и $m2$ имеют один и тот же хеш-код h = hash($m1$) = hash($m2$), где hash() – используемый алгоритм хеширования [242, 246].

Предотвратить существование коллизий принципиально невозможно, поскольку для n-битового алгоритма хеширования размер множества возможных хеш-кодов составляет 2^n, тогда как размер множества хешируемых сообщений является бесконечным. Трудоемкость нахождения коллизий является одной из основных характеристик криптостойкости хеш-функций: для стойкого алгоритма хеширования поиск коллизии должен требовать порядка $2^{n/2}$ операций хеширования.

Существует также понятие мультиколлизии (multicollision) – множества из r сообщений, каждое из которых при хешировании дает один и тот же результат [153, 188]. В работе [153] показано, что трудоемкость поиска мультиколлизии лишь незначительно (логарифмически) превышает трудоемкость поиска коллизии для того же алгоритма хеширования. Мультиколлизии будут подробно описаны далее.

Нахождение прообраза

Атака, направленная на поиск прообраза (preimage attack), ставит своей целью нахождение сообщения, хеш-код которого соответствует заданному хеш-коду h. Существуют два типа таких атак [242, 246]:

- поиск первого прообраза (first preimage attack) – поиск сообщения m, для которого hash(m) = h;
- поиск второго прообраза (second preimage attack) – при имеющемся сообщении $m1$ поиск другого сообщения $m2$, удовлетворяющего условию: hash($m2$) = hash($m1$).

Считается, что алгоритм хеширования является стойким против данных атак, если для поиска прообраза необходимо не менее 2^n операций хеширования алгоритмом, вычисляющим n-битовые хеш-коды.

Атаки, направленные на поиск прообраза, существенно более опасны, чем атаки, ставящие своей целью поиск коллизий, поскольку они могут быть использованы, в частности, для следующих целей [166]:

- компрометации схем проверки целостности;
- подделки электронной подписи;
- поиска паролей по их известным хеш-кодам.

Определение секретного ключа

Определение секретного ключа как цель атаки актуально для ключевых алгоритмов хеширования или ключевых надстроек над бесключевыми алгоритмами хеширования.

Промежуточные цели атак на алгоритмы хеширования

Нахождение near-коллизии

Near-коллизией называется пара сообщений, хеш-коды которых являются почти эквивалентными, с различиями только в нескольких битах [72].

Само по себе обнаружение near-коллизии не может быть опасным. Однако поиск near-коллизий может использоваться в контексте различных атак на хеш-функции. В качестве примера использования near-коллизии для поиска реальной коллизии можно привести пример атаки на алгоритм SHA, направленной на поиск многоблочной коллизии и описанной в [73]. Поэтому считается, что криптографически стойкие хеш-функции должны также не быть подверженными поиску near-коллизий [174].

Нахождение псевдопрообраза

Псевдопрообраз (pseudo-preimage) – это сообщение *m*, для которого hash(*IV'*, *m*) = *h*, где:

- *h* – заданный хеш-код;
- *IV'* – начальный хеш-код, использованный при вычислении хеш-кода *h*.

Таким образом, псевдопрообраз отличается от первого прообраза тем, что при его поиске не фиксируется начальный хеш-код, т. е. может быть использовано другое начальное значение, нежели прописанное в алгоритме хеширования (например, для алгоритма SHA начальное значение – это совокупность исходных значений регистров *A*...*E* – см. описание алгоритма SHA в разделе 1.3). Значение *IV'*, используемое для поиска псевдопрообраза, может иметь различные специфические свойства, существенно упрощающие поиск псевдопрообраза по сравнению с поиском прообраза (при стандартном начальном значении) [166].

Как и near-коллизии, псевдопрообразы сами по себе не являются опасными (если не рассматривать заведомо ошибочные варианты реализации алгоритмов хеширования, позволяющие изменять или загружать начальные значения), однако псевдопрообразы могут являться промежуточным шагом других атак, в частности они могут быть успешно использованы в контексте атак, посвященных поиску прообраза ([103, 166]).

Нахождение псевдоколлизии

По аналогии с псевдопрообразом, псевдоколлизия определяется как ситуация, в которой два различных сообщения *m*1 и *m*2 имеют один и тот же хеш-код *h* = hash(*m*1, IV_1) = hash(*m*2, IV_2), но, в отличие от «классической» коллизии, псевдоколлизия возникает при различных начальных значениях алгоритма хеширования: IV_1 и IV_2.

В работе [250] утверждается, что криптографические алгоритмы хеширования должны быть стойкими к поиску псевдоколлизий (т. е. трудоемкость поиска псевдоколлизии не должна быть ниже $2^{n/2}$ операций для *n*-битового алгоритма хеширования), поскольку нахождение псевдоколлизии также может являться промежуточным шагом, снижающим итоговую трудоемкость различных атак на алгоритмы хеширования.

Рассмотрим далее различные методы криптоанализа алгоритмов хеширования и атаки на них, приводящие к достижению описанных выше целей.

1.2.2 Атаки методом «грубой силы»

Метод «грубой силы» (brute-force attack) предполагает перебор всех возможных вариантов каких-либо объектов (в частности, хешируемых сообщений) до нахождения искомого значения (например, сообщения, соответствующего заданному хеш-коду).

Атаки методом «грубой силы» могут быть применены для достижения всех возможных целей атак на алгоритмы хеширования:

- поиска коллизии;
- поиска прообраза;
- определения секретного ключа (для ключевых алгоритмов хеширования).

Рассмотрим последнюю из рассматриваемых целей. Пусть размер секретного ключа в битах равен b. Соответственно, существует 2^b вариантов ключа. Криптоаналитик должен методично перебрать все возможные ключи, т. е. (если рассматривать b-битовую последовательность как число) применить в качестве ключа значение 0, затем 1, 2, 3 и т. д. до максимально возможного (2^b – 1). В результате секретный ключ обязательно будет найден, причем в среднем такой поиск потребует $2^b/2$, т. е. 2^{b-1} тестовых операций [34].

В качестве критерия корректности найденного ключа используется N пар сообщений и их хеш-кодов. Поскольку при переборе ключей возможны коллизии, в [202] показано, что требуемое число пар для определения верного ключа «несколько превышает» (slightly larger) значение b/n для n-битового алгоритма хеширования.

Защита от атак методом «грубой силы» весьма проста – достаточно лишь увеличить размер ключа: увеличение размера ключа на 1 бит увеличит количество ключей (и среднее время атаки) в 2 раза.

Существуют различные методы усиления эффективности атаки методом «грубой силы», в частности [84]:

- атака методом «грубой силы» простейшим образом распараллеливается: при наличии, скажем, миллиона компьютеров, участвующих в атаке, ключевое множество делится на миллион равных фрагментов, которые распределяются между участниками атаки;
- скорость перебора ключей может быть во много раз увеличена, если в переборе участвуют не компьютеры общего назначения, а специализированные устройства.

Следует учесть, что с развитием вычислительной техники требования к размеру секретного ключа постоянно возрастают. В качестве примера (применительно к симметричному шифрованию) можно привести тот факт, что в той же работе [84] был рекомендован 90-битовый размер ключа в качестве абсолютно безопасного (причем с 20-летним запасом) на конец 1995 г. В 2000 г. эксперты посчитали безопасным с примерно 80-летним запасом использование ключей размером от 128 бит [101].

В известной работе Даниэля Бернштейна (Daniel J. Bernstein) [66] указано, что, несмотря на всю силу параллельных атак методом «грубой силы», криптографы уделяют данной проблеме достаточно мало внимания, допуская следующие ошибки при проектировании криптографических алгоритмов и протоколов и их реализации:

- криптографы часто сильно преувеличивают реальную стойкость своих криптосистем из-за того, что рассматривают только последовательные (т. е. использующие один атакующий компьютер) атаки на криптосистему, не уделяя внимания более сильным параллельным атакам;
- в случае если криптосистема архитектурно зависима от решения, параллельным или последовательным атакам она должна противостоять, часто делается неверный выбор в пользу последовательных атак.

Атака методом «грубой силы» является «мерилом эффективности» других атак – атака считается тем эффективнее, чем быстрее она достигает требуемой цели по сравнению с атакой методом «грубой силы».

1.2.3 Словарные атаки и цепочки хеш-кодов

Словарная атака (dictionary attack) – это метод вскрытия какой-либо информации путем перебора возможных значений данной информации (здесь и далее для определенности в качестве мишени словарных атак будем рассматривать пароли пользователей, хранящиеся в виде их хеш-кодов; при этом область применения словарных атак достаточно широка) [246].

Название атаки произошло благодаря тому, что основу множества перебираемых паролей изначально составляли слова какого-либо языка, а словарные атаки эксплуатировали присущую большинству пользователей тенденцию к использованию легко запоминаемых паролей, к которым часто относятся различные слова и их варианты (например, замена части букв слова на похожие по написанию цифры или спецсимволы).

Словарные атаки часто классифицируют как один из вариантов атаки методом «грубой силы», поскольку здесь также выполняется перебор вариантов пароля или ключа. По сравнению с методом «грубой силы» словарные атаки осуществляют перебор по существенно меньшему (но наиболее вероятному ввиду того, что пользователи нередко выбирают простые и легко запоминающиеся пароли) множеству возможных значений.

Далее рассмотрим некоторые варианты словарных атак и методы противодействия им.

Словарная атака, основанная на предварительных вычислениях

Смысл данного варианта словарной атаки состоит в предварительном вычислении хеш-кодов паролей, принадлежащих какому-либо множеству.

Путем такого вычисления атакующий получает таблицу соответствий входящих в множество паролей и их хеш-кодов (см. рис. 1.9). Для нахождения искомого пароля, соответствующего известному злоумышленнику хеш-коду h, выполняется поиск хеш-кодов в таблице, в результате которого делается один из следующих выводов:

- если в таблице нашлась запись, соответствующая искомому значению h, то пароль p (для которого $h = \text{hash}(p)$) найден; при этом стоит учитывать, что поскольку возможно возникновение коллизий, найденный пароль может быть не искомым паролем, а паролем с эквивалентным искомому хеш-кодом;

- если записи со значением h в таблице нет, то искомый пароль не принадлежит к множеству, покрываемому данной таблицей; можно продолжить поиск с помощью других таблиц или с применением иных методов.

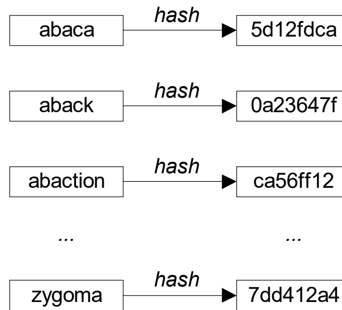

Рисунок 1.9. Таблица паролей и хеш-кодов

Основным недостатком данной атаки является достаточно большой объем памяти, требуемой на хранение таблицы, размер которого в битах n-битового алгоритма хеширования можно оценить следующим образом:

$$\mathrm{V}(T) = (n + k) * |P|,$$

где:

- P – множество паролей, покрываемое таблицей T;
- k – средний размер пароля в битах.

Цепочки хеш-кодов

Использование цепочек хеш-кодов (hash chains) – это улучшенный метод словарной атаки. Улучшение состоит в том, что при использовании цепочек на хранение таблицы паролей и соответствующих им хеш-кодов требуется существенно меньше памяти, чем в случае классической словарной атаки.

Принцип действия цепочек хеш-кодов основан на введении дополнительной функции R(), с помощью которой любому хеш-коду h можно псевдослучайным образом сопоставить некий пароль p из множества паролей P [123, 144]:

$$p = \mathrm{R}(h).$$

С помощью функции R() и функции хеширования hash() можно сформировать цепочки из паролей $p_1 \ldots p_N$ и соответствующих им хеш-кодов $h_1 \ldots h_N$:

$$p_1 \to h_1 \to p_2 \to h_2 \to \ldots \to p_N \to h_N. \quad (1.1)$$

Простой пример функции R() приведен в [147]: если множеством паролей является множество 6-значных десятичных чисел, то выходным значением функции R() могут быть первые 6 цифр хеш-кода в десятичном представлении.

Исходные значения p_1 для каждой цепочки могут выбираться по любому принципу, в том числе генерироваться случайным образом в пределах покрываемого таблицей множества паролей.

Атакующий объединяет вычисленные таким образом цепочки в таблицу (см. рис. 1.10). Принципиальным моментом является то, что для уменьшения

требуемой для хранения памяти сохраняется не вся таблица, а лишь первое и последнее значения (т. е. p_1 и h_N) для каждой строки.

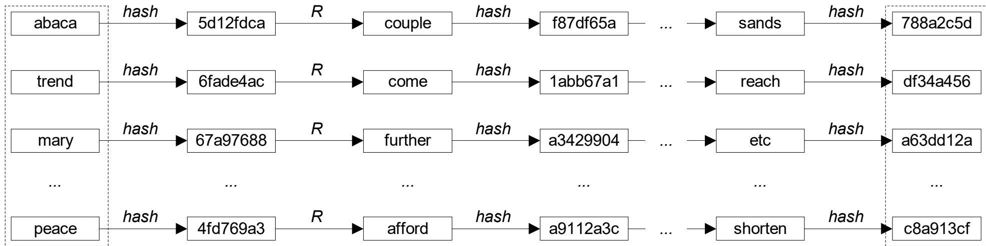

Рисунок 1.10. Таблица цепочек хеш-кодов

При необходимости найти пароль, соответствующий заданному хеш-коду h_x, атакующий вычисляет следующую цепочку:

$$h_x \to p_{x+1} \to h_{x+1} \to \ldots \to p_{x+N-1} \to h_{x+N-1}. \qquad (1.2)$$

Каждое из получаемых в процессе данных вычислений значений h_{x+i} сравнивается с хранящимися в таблице значениями h_N каждой строки. Если на каком-либо этапе обнаружены эквивалентные значения h_{x+i} и h_N, то дальнейшие вычисления цепочки (1.2) не выполняются, а поиск требуемого пароля выполняется восстановлением данной строки таблицы, т. е. вычислением цепочки (1.1) до нахождения некоторого значения h_j, равного h_x. При нахождении такого значения искомый пароль определяется как значение p_j данной цепочки.

Поскольку возможно возникновение коллизий, восстановление строки таблицы может не приводить к нахождению требуемого значения h_j (см. пример на рис. 1.11; данный пример показывает также то, что коллизии уменьшают покрываемое таблицей множество паролей). В этом случае атакующий продолжает вычисления цепочки (1.2) до нахождения следующего совпадения h_{x+i} и h_N. Если вычисления цепочки (1.2) завершены, а совпадения не найдены, то атакующим делается вывод о том, что данная таблица не содержит искомый пароль.

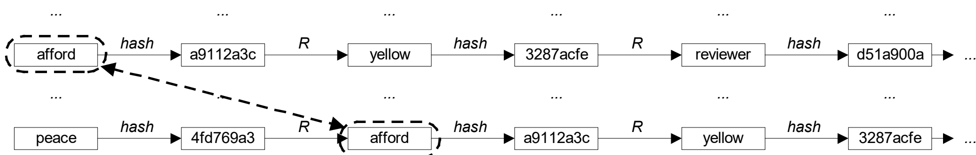

Рисунок 1.11. Пример коллизии в цепочке хеш-кодов

Значение N является параметром таблицы цепочек, от которого зависят две важные характеристики таблицы:

- размер памяти для хранения таблицы цепочек обратно пропорционален значению N при эквивалентном покрытии множества паролей (без учета коллизий, вероятность которых нелинейно возрастает с ростом N);
- сложность нахождения требуемого пароля в таблице, которая возрастает с ростом N.

Использование цепочек хеш-кодов уменьшенной длины

Как показано выше (см. рис. 1.11), возникновение коллизий может приводить к частичному совмещению строк таблицы, что, в свою очередь, приводит к уменьшению покрытия множества паролей при заданных размерах таблицы [191, 246].

Вторая проблема – это «зацикленные» строки, возникающие из-за того, что в процессе вычисления строки произошло совпадение паролей (текущего и одного из предыдущих) или хеш-кодов (т. е. для неких i и j какой-либо одной строки $p_i = p_j$ или $h_i = h_j$). Стоит отметить, что «зацикленные» строки могут быть обнаружены на этапе вычислений и отброшены.

Использование нескольких таблиц меньшего размера вместо одной большой таблицы – это одно из направлений усиления таблиц цепочек хеш-кодов. В каждой из таких таблиц используется своя функция преобразования из хеш-кода в пароль, т. е. в совокупности таблиц используются несколько функций $R_1()...R_T()$ (где T – количество таблиц).

В этом случае при возникновении коллизий в разных таблицах не происходит дальнейшего совмещения строк, поскольку в разных таблицах используются различные функции R(). Коллизия в одной таблице приведет к совмещению строк, но вероятность такой коллизии в T раз меньше по сравнению с коллизией в разных таблицах.

Использование цепочек хеш-кодов переменной длины

Второе направление усиления таблиц цепочек хеш-кодов – это использование строк переменной длины (но не превышающей некоего порогового значения L) [88]. Строка таблицы завершается при вычислении некоего специфического значения h_i. Пример такого специфического значения – хеш-код, у которого значение первых двух байт равно нулю [147] (см. рис. 1.12). Следует, однако, учесть, что пороговое значение L должно быть достаточно большим, чтобы в подавляющем большинстве строк специфический хеш-код достигался. Если же такое значение после вычисления L хеш-кодов не достигается, то вычисляемая строка отбрасывается, поскольку считается «зацикленной».

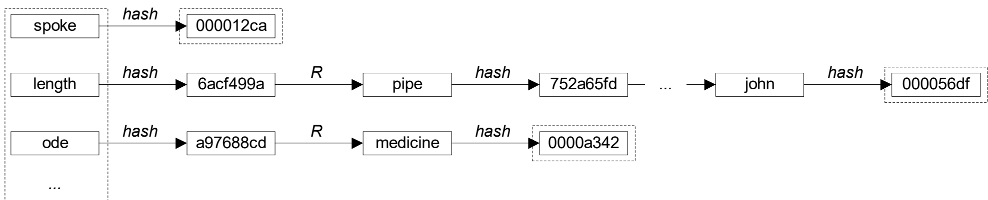

Рисунок 1.12. Таблица цепочек переменной длины

Данный вид таблиц помогает и против совмещения строк при коллизиях, поскольку в этом случае обе строки приведут к одному и тому же специфическому значению. Таким образом, совмещенные строки легко обнаруживаются, после чего одну из таких строк (менее длинную) можно отбросить и пересчитать с другим начальным значением.

Направление применения таблиц цепочек хеш-кодов активно развивается (см., например, [146, 228]) наряду с радужными таблицами, подробно описанными далее.

1.2.4 Радужные таблицы

Радужная таблица (rainbow table) – это одно из направлений усиления цепочек хеш-кодов с точки зрения повышения эффективности их применения.

В радужных таблицах также используется последовательность функций преобразования из хеш-кода в пароль, но каждая функция $R_i()$ используется для преобразования i-го хеш-кода h_i в $(i + 1)$-й пароль p_{i+1}. Таким образом, в таблице последовательно применяются функции $R_1()...R_{N-1}()$ (см. рис. 1.13).

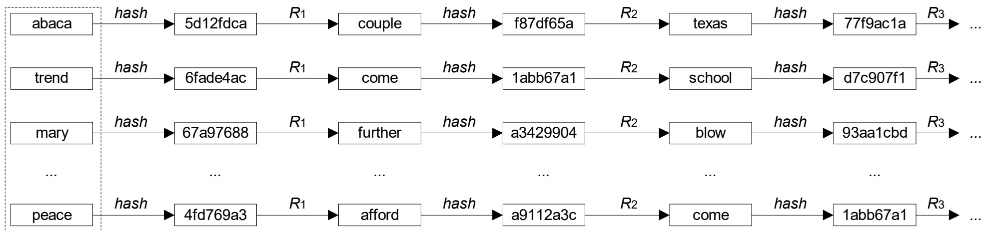

Рисунок 1.13. Радужная таблица

Радужные таблицы обеспечивают более высокую скорость поиска паролей по сравнению с описанными выше вариантами при эквивалентных размерах (а также несколько более высокий процент покрытия множества паролей).

Две описанные выше проблемы таблиц цепочек хеш-кодов решаются радужными таблицами достаточно элегантно [147, 191, 246]:

- «зацикленные» строки невозможны в принципе, поскольку в любой строке используется вся последовательность функций преобразования;
- коллизия приводит к совмещению строк только в случае, если она возникает в одном столбце (вероятность чего в N раз меньше вероятности коллизии во всей таблице); следовательно, значения h_N у таких строк становятся эквивалентными – лишнюю строку можно отбросить и пересчитать с другим начальным значением.

Поиск требуемого пароля в радужной таблице отличается от описанных выше вариантов; он выполняется следующим образом:

1. Сначала ищется совпадение заданного хеш-кода h_x с каким-либо из значений h_N таблицы.
2. Если совпадение не обнаружено, то предполагается, что h_x должно совпадать с каким-либо из h_{N-1}. Следовательно, вычисляется

   $$h_{x+1} = \mathrm{hash}(R_{N-1}(h_x))$$

   и сравнивается со значениями h_N.
3. Если совпадение снова не найдено, предположение о местонахождении h_x в таблице «смещается» еще на один столбец влево, т. е. с h_N сравнивается уже следующее значение:

$$h_{x+2} = \text{hash}(R_{N-1}(\text{hash}(R_{N-2}(h_x)))).$$

4. И так далее, до нахождения соответствия. Если соответствие найдено, то расчет искомого пароля выполняется с начала строки, как во всех описанных выше вариантах. Если же соответствие не найдено до завершения обработки предположения, что $h_x = h_1$, то считается, что искомый пароль в данной таблице отсутствует.

Название радужных таблиц произошло от аналогии каждой функции $R_i()$ с каким-либо цветом, тогда таблица с длинными тонкими разноцветными полосками будет напоминать радугу [147]. Радужные таблицы активно используются для атак на реальные криптосистемы – см., например, [134, 176, 192, 206]. Кроме того, рассматриваются варианты комбинирования радужных таблиц с таблицами цепочек переменной длины, использующими специфические значения [146].

Стоит отметить, что радужные таблицы и описанные в предыдущем разделе словарные атаки изначально предлагались к использованию для нахождения ключей симметричного шифрования, т. е. в качестве паролей (например, в радужной таблице) фигурируют ключи, а в качестве хеш-кодов – результаты зашифрования константного блока открытого текста на конкретном ключе. Следовательно, функция R() в данном случае преобразует блок шифртекста в некий возможный ключ.

1.2.5 Парадокс «дней рождения» и поиск коллизий

Парадокс «дней рождения» состоит в том, что для получения из множества, содержащего N видов элементов, двух элементов одинакового вида требуется сделать $O(\sqrt{N})$ попыток (поэтому атаки, использующие парадокс «дней рождения», называют также «атаками квадратного корня» [246]). Например, в среднестатистической группе из 23 человек у двух человек совпадут дни рождения с вероятностью, превышающей 50 %, что выглядит несколько удивительным (именно от этого примера произошло название парадокса «дней рождения») [46]. Данный парадокс активно используется в криптоанализе.

Именно благодаря парадоксу «дней рождения» атака методом «грубой силы» для поиска коллизии требует в $2^{n/2}$ раз меньше операций, чем для поиска прообраза.

Парадокс «дней рождения» может быть напрямую использован для атак на хеш-функции, поскольку от него зависит трудоемкость, требуемая для нахождения коллизии. В [154] и [202] приведен пример следующей атаки, которую может выполнить злоумышленник (данную атаку предложил Гидеон Юваль (Gideon Yuval) еще в 1979 г.):

1. Злоумышленнику необходимо выдать поддельный документ за действительный документ, подписанный неким пользователем системы.
2. Злоумышленник генерирует r вариантов поддельного документа и столько же вариантов оригинального документа (и в том, и в другом случае под «вариантами» подразумеваются документы, идентичные по смыслу, но различные в каких-либо деталях, незначительных в контексте документа, но приводящих к различным хеш-кодам). Значение r может быть вычислено с помощью парадокса «дней рождения»: например, $2^{n/2}$ для n-битового алгоритма хеширования.

3. Злоумышленник выполняет поиск коллизий между оригинальными и поддельными документами. Если коллизия найдена, т. е. найдены оригинальный документ m_x и поддельный документ f_y, такие, что $\text{hash}(m_x) = \text{hash}(f_y)$, то злоумышленник предлагает пользователю подписать именно вариант оригинального документа m_x.
4. Электронная подпись пользователя вместо документа m_x прикрепляется злоумышленником к документу f_y. Подпись пользователя под подложным документом будет верна именно благодаря найденной злоумышленником коллизии.

Из-за наличия парадокса «дней рождения» выходное значение алгоритмов хеширования должно быть достаточно большим – его размер в битах не менее, чем в два раза, должен превышать размер, достаточный для успешного противодействия атакам методом «грубой силы», направленным на поиск коллизии.

Весьма интересную особенность парадокса «дней рождения» отметили авторы работы [63] Михир Белэр (Mihir Bellare) и Тадайоши Коно (Tadayoshi Kohno). Они ввели понятие «регулярности» (amount of regularity) хеш-функций, которое фактически означает степень равномерности распределения возможных сообщений по множеству выходных значений алгоритма хеширования. Приведенные выше оценки трудоемкости поиска коллизий справедливы только для регулярных хеш-функций (т. е., другими словами, функций, в которых любые возможные хеш-коды встречаются одинаково часто). Для хеш-функций, которые не отличаются регулярностью, трудоемкость поиска коллизии может быть значительно ниже. Авторы данной работы предложили методики количественной оценки регулярности алгоритмов хеширования и оценки нижней границы трудоемкости поиска коллизии в зависимости от регулярности.

В работе [202] Барт Пренель (Bart Preneel) выполнил анализ требуемого размера выходного значения хеш-функций, достаточного для противостояния атакам, использующим парадокс «дней рождения», в том числе для противодействия злоумышленникам, обладающим «огромным» (до 64 терабайт) объемом памяти и большим количеством атакующих рабочих станций. В результате автор данной работы в 2003 г. делает вывод, что 160-битового хеш-кода вполне достаточно с запасом, как минимум, на 20 лет вперед. В настоящее время данное значение уже не выглядит однозначно достаточным.

Для оценки криптостойкости алгоритмов хеширования криптоаналитики часто используют различные методы поиска коллизий (в отличие от описанного выше метода, в данном случае достаточно найденной коллизии у незначащих (абстрактных) сообщений) и сравнивают криптостойкость алгоритма с указанной выше теоретической – $2^{n/2}$ операций для n-битового алгоритма хеширования.

«Классический» поиск коллизий выполняется путем выбора некоторых базовых значений хешируемых сообщений $m_0 \dots m_N$, над которыми выполняется вычисление цепочек хеш-кодов:

$$m_i \to \text{hash}(m_i) \to \text{hash}(\text{hash}(m_i)) \to \dots$$

Все значения каждой из цепочек записываются и сравниваются с уже записанными. Совпадение значений сигнализирует о нахождении коллизии.

Такой подход требует весьма серьезных ресурсов памяти для хранения всех промежуточных результатов. В 1987 г. авторы работы [205] предложили усовершенствование данного метода, заключающееся в том, что хранению в памяти подлежат не все промежуточные результаты вычислений, а только те из них, которые имеют специфические значения (аналогично использованию специфических значений в описанных выше словарных атаках); в качестве примера специфических значений авторы [205] приводят значения с 20 нулевыми битами слева. При этом только специфические значения сохраняются и сравниваются (остальные значения отбрасываются), что на несколько порядков снижает требования данного метода к памяти.

Как и при классическом подходе, совпадение каких-либо специфических значений сигнализирует о нахождении коллизии. Аналогично вычислению цепочек хеш-кодов переменной длины в словарных атаках, в данном методе поиска коллизий также устанавливается некое (весьма большое) пороговое значение максимальной длины цепочки, при достижении которого цепочка считается зацикленной и отбрасывается.

Несмотря на существенно сниженные требования к ресурсам метода со специфическими значениями (по сравнению с классическим подходом), авторы работы [240] в 1994 г. утверждали, что «существующая технология поиска коллизий мало применима на практике, пока она не может эффективно распараллеливаться». Они предложили эффективный метод распараллеливания поиска коллизий, суть которого в следующем (см. рис. 1.14 из [240]):

- каждый участвующий в поиске компьютер обрабатывает свою цепочку хеш-кодов, начиная с некоторого значения m_i;
- при достижении специфического значения компьютер сообщает о нем остальным участникам поиска (или в некий центр, управляющий поиском);
- другие участники поиска (или управляющий центр) сравнивают специфические значения с вычисленными на более ранних этапах; совпадение свидетельствует о коллизии.

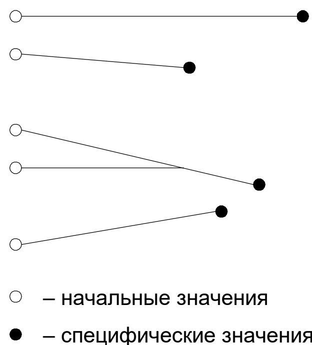

Рисунок 1.14. Параллельный поиск коллизий

Данный метод распараллеливания был испытан авторами на практике для поиска коллизий в алгоритме хеширования MD5 [212], который будет рассмотрен далее в разделе 1.3.

Отдельно рассмотрим мультиколлизии, которые могут быть эффективными при проведении некоторых атак на алгоритмы хеширования и трудоемкость поиска которых только незначительно превышает трудоемкость поиска обычных коллизий.

Мультиколлизии были предложены в 2004 г. в работе [153]. Опишем мультиколлизии, для чего введем следующие обозначения:

- f(a, b) – функция сжатия алгоритма хеширования, где a – предыдущее значение переменных состояния, а b – обрабатываемый блок данных; предполагается, что функция сжатия является слабой, т. е. в данном случае существует эффективный метод поиска коллизий для этой функции сжатия;
- h_i – значение переменных состояния на i-й итерации (см. далее);
- B_i и B_i' – блоки сообщений, дающие коллизию на i-й итерации;
- t – количество итераций (определяется атакующим исходя из требований атаки).

Тогда алгоритм поиска мультиколлизий выглядит следующим образом:

1. В качестве начального значения переменных состояния h_0 используется стандартный вектор инициализации алгоритма.
2. В цикле по i от 1 до t выполняются следующие действия:
- определяется значение блоков B_i и B_i', дающих коллизию функции сжатия:

$$f(h_{i-1}, B_i) = f(h_{i-1}, B_i');$$

- текущим значением переменных состояния становится значение f(h_{i-1}, B_i).

3. После выполнения цикла атакующий получает мультиколлизию – 2^t сообщений с эквивалентным хеш-кодом (рис. 1.15), каждое из которых имеет следующий вид:

$$(b_1, \ldots, b_t, pad),$$

где b_i – это одно из значений B_i и B_i', а pad – стандартное дополнение сообщения.

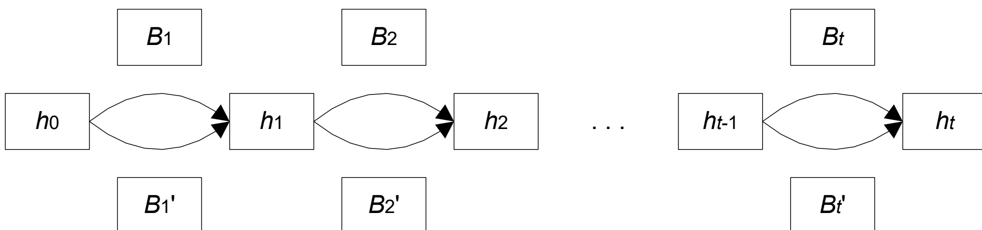

Рисунок 1.15. Мультиколлизия

1.2.6 Дифференциальный криптоанализ

Данный метод атак был изобретен в 1990 г. известными израильскими криптологами Эли Бихамом (Eli Biham) и Ади Шамиром (Adi Shamir) и опубликован в ряде их работ (см., например, [74, 75]). Однако не менее известный криптолог

Брюс Шнайер (Bruce Schneier) в своей книге [53] утверждает, что дифференциальный криптоанализ был открыт существенно раньше, но не появлялся в открытой печати. Тем не менее именно Бихам и Шамир до сих пор считаются изобретателями дифференциального криптоанализа.

Дифференциальный криптоанализ в равной степени применим как против алгоритмов симметричного шифрования (изначально дифференциальный криптоанализ был предложен для атаки на стандарт шифрования США DES (Data Encryption Standard – стандарт шифрования данных) [124]), так и против алгоритмов хеширования.

Рассмотрим дифференциальный криптоанализ в применении к алгоритмам хеширования на примере криптоанализа алгоритма SHI1, выполненного в работе [93].

SHI1 является упрощенным вариантом алгоритма хеширования SHA (подробное описание алгоритма SHA будет приведено в разделе 1.3), в итерации которого все нелинейные элементы заменены на побитовую логическую операцию «исключающее или» (XOR).

Таким образом, итерация алгоритма SHI1 выглядит следующим образом (рис. 1.16):

$$t = (a <<< 5) \oplus b \oplus c \oplus d \oplus e \oplus W_i \oplus K_i\ ;$$

$$e = d;$$

$$d = c;$$

$$c = b <<< 30;$$

$$b = a;$$

$$a = t,$$

где:

- $a \ldots e$ – 32-битовые переменные, впоследствии модифицирующие текущее состояние алгоритма (см. описание алгоритма SHA);
- t – временная 32-битовая переменная;
- <<< – операция циклического сдвига (вращения) влево на указанное число битов;
- K_i – 32-битовые модифицирующие константы;
- W_i – 32-битовый фрагмент расширенного блока сообщения.

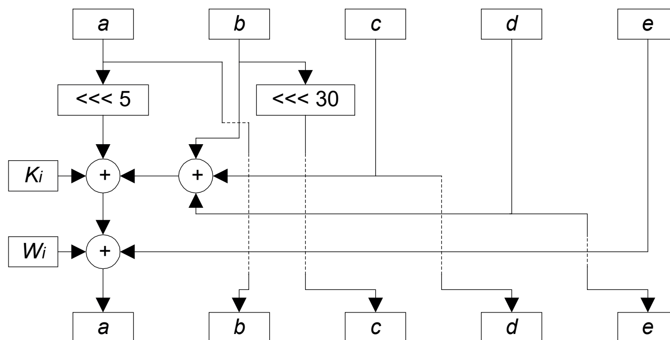

Рисунок 1.16. Итерация алгоритма SHI1

Дифференциальный криптоанализ основан на анализе пар сообщений, между которыми существует определенная разность (difference). Для алгоритма SHI1 авторы работы [93] в качестве разности Δ фрагмента расширенного блока сообщения рассматривают результат операции XOR между данными фрагментами:

$$\Delta = W1_i \oplus W2_i,$$

где $W1_i$ и $W2_i$ – соответственно i-е фрагменты расширенных блоков первого и второго сообщений.

При анализе различных алгоритмов разность текстов может быть определена различным образом, например при анализе алгоритма MD4 (см. раздел 1.3) в работе [244] разность определена авторами работы как результат вычитания 32-битовых величин по модулю 2^{32}.

Авторы атаки на алгоритм SHI1 рассмотрели ситуацию, когда $W1_i$ и $W2_i$ отличаются только в одном бите, а именно в бите № 1 (считая, что биты данных величин нумеруются справа налево, начиная с 0); для определенности будем считать, что бит № 1 $W1_i$ имеет значение 0, а бит № 1 $W2_i$ – значение 1. Данное отличие в бите № 1 в последующих итерациях обработки блоков $W1$ и $W2$ приведет к следующим изменениям (рис. 1.17):

- будут различаться биты № 1 у переменной a в итерации $i + 1$;
- будут различаться биты № 1 у переменной b в итерации $i + 2$ (поскольку в итерации присутствует операция $b = a$);
- кроме того, переменная a участвует в вычислении t, которое повлечет за собой дальнейшие нежелательные различия; однако эти различия можно скорректировать, если сделать значения $W1_{i+1}$ и $W2_{i+1}$ различающимися в бите № 6 (поскольку операция (a <<< 5) переместит различающиеся биты № 6 в биты № 1); здесь и далее корректирующие биты различных фрагментов $W1$ и $W2$ должны принимать значения 0 для $W1$ и 1 для $W2$ (а не наоборот), в этом случае операция XOR сделает данные биты нулевыми и в $W1$, и в $W2$;
- будут различаться биты № 31 у переменной c в итерации $i + 3$ (благодаря операции $c = b$ <<< 30 различающиеся биты сдвинутся в позицию № 31);
- переменная b (у которой различаются биты № 1) также участвует в вычислении t; следовательно, для коррекции распространения различий нужно использовать значения $W1_{i+2}$ и $W2_{i+2}$, различающиеся в бите № 1;
- в следующей итерации ($i + 4$) различающийся бит № 31 переменной c переместится в бит № 31 переменной d, а нежелательные изменения из-за зависимости t от c скорректируются различием в бите № 31 переменных $W1_{i+3}$ и $W2_{i+3}$;
- аналогичным образом в итерации ($i + 5$) будут различаться биты № 31 переменной e, а значения $W1_{i+4}$ и $W2_{i+4}$ для коррекции различий в d необходимо использовать с различными битами № 31;
- и наконец, для коррекции различающихся битов № 31 в переменной e используются $W1_{i+5}$ и $W2_{i+5}$ с различными битами № 31.

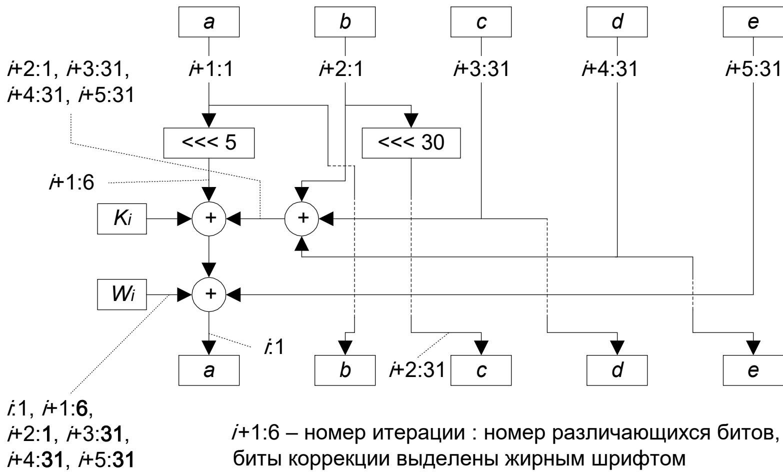

Рисунок 1.17. Распространение и коррекция разности в бите № 1 *i*-х фрагментов расширенных блоков алгоритма SHI1

Таким образом, различия в бите № 1 *i*-х фрагментов расширенных блоков сообщений будут скомпенсированы различиями в определенных битах последующих пяти фрагментов. При этом значения переменных *a*...*e* по прошествии данной последовательности итераций не изменятся. Следовательно, мы имеем два блока разных хешируемых сообщений с одинаковым влиянием на результирующие хеш-коды, но при этом с различиями в следующих битах:

Таблица 1.1. Различающиеся биты фрагментов расширенных блоков при дифференциальном криптоанализе алгоритма SHI1

№ различающихся фрагментов расширенных блоков	№ различающихся битов
i	1
$i + 1$	6
$i + 2$	1
$i + 3$	31
$i + 4$	31
$i + 5$	31

Такую ситуацию авторы атаки назвали «локальной коллизией» (local collision). Следует учесть, что данное отличие в бите № 1 не должно возникать после 75-й итерации алгоритма, поскольку для коррекции необходимо еще 5 итераций, а всего в алгоритме SHA выполняется 80 итераций. Кроме того, авторы атаки утверждают, что бит № 1 выбран исключительно для удобства иллюстрации атаки – вместо данного бита может быть использован любой другой бит $W1_i$ и $W2_i$.

Следующий этап атаки – распространение данной локальной коллизии на полнораундовый алгоритм, включая как остальные итерации, так и процедуру расширения хешируемого блока. Далее (в описании алгоритма SHA) приведена процедура расширения 512-битового хешируемого блока на 80 32-битовых значений $W_0...W_{79}$. Именно эта процедура используется для конструирования вектора возмущений (disturbance vector фактически представляет собой таблицу размером 80×32 (по числу битов в $W_0...W_{79}$), показывающую, какие биты расширенных блоков необходимо определенным образом модифицировать для достижения локальной коллизии [96]) и, на его основе, исходных сообщений, вызывающих коллизию. Подробный пример конструирования вектора возмущений и вызывающих коллизию сообщений приведен в [93].

Таким образом, результатом атаки является множество сообщений (два – частный случай), при вычислении хеш-кодов которых возникает коллизия.

Простота описанной выше атаки методом дифференциального криптоанализа на алгоритм SHI1 объясняется отсутствием в его итерации нелинейных операций.

Авторы работы [93] рассмотрели еще две упрощенные модификации алгоритма SHA, в которых в итерации SHI1 вводились нелинейные операции – сложение по модулю 2^{32} или функции $f_i()$ (их описание приведено далее). В обоих этих случаях атака на данные алгоритмы развивается аналогично SHI1, однако с одним исключением: в описанные ранее варианты распространения изменений вносится определенная вероятность благодаря нелинейным функциям.

Вероятность в данном случае приводит к тому, что, по сравнению с поиском коллизии для SHI1 появляется множество «ложных» коллизий, т. е. сообщений, удовлетворяющих выработанным в рамках атаки критериям, но с различающимися хеш-кодами. Трудоемкость атаки становится несравнимо выше именно из-за необходимости расчета хеш-кодов по всему множеству данных сообщений с целью поиска реальных коллизий.

В качестве примера использования дифференциального криптоанализа против реального алгоритма хеширования можно привести описанную в той же работе [93] атаку на полнораундовый алгоритм SHA, трудоемкость которой (поиск коллизии) составляет 2^{61} операций.

1.2.7 Алгебраический криптоанализ

Алгебраическим криптоанализом называется метод анализа, который использует алгебраические свойства анализируемого алгоритма [100].

Данный метод анализа в настоящее время активно используется против алгоритмов блочного симметричного шифрования. В частности, известно достаточно много работ, посвященных алгебраическому анализу стандарта шифрования AES (Advanced Encryption Standard – «улучшенный стандарт шифрования») [131]; в качестве примеров таких работ можно привести работы [122, 135, 186].

В рамках алгебраического криптоанализа блочный шифр представляется в виде системы уравнений относительно значений битов ключа, решение ко-

торых находит искомый ключ или его фрагменты. Возможно использование алгебраического криптоанализа и в контексте других атак.

Есть примеры использования алгебраического криптоанализа и против алгоритмов хеширования, например Макото Сугита (Makoto Sugita), Мицуру Кавазое (Mitsuru Kawazoe) и Хидеки Имаи (Hideki Imai) в 2006 г. успешно применили в комплексе алгебраический и дифференциальный криптоанализы для атаки на алгоритм SHA-1 с уменьшенным количеством итераций [231].

1.2.8 Атаки, использующие утечки данных по побочным каналам

В процессе своей работы программная или аппаратная реализация алгоритма хеширования может допускать различные утечки информации об обрабатываемых данных и/или секретном ключе (если используется ключевой алгоритм хеширования или соответствующая надстройка). Такими утечками могут быть, в частности, следующие (подробнее см., например, [34]):

- электромагнитное излучение;
- информация о затратах времени и мощности на обработку данных;
- информация об ошибках, возникающих в процессе хеширования, и т. д.

Данную информацию криптоаналитик может использовать при атаке на реализацию алгоритма хеширования, а также в контексте других атак. Такие атаки называются атаками, использующими утечки данных по побочным каналам (side-channel attacks). Стоит, однако, сказать, что более часто эти атаки применяются против алгоритмов симметричного шифрования.

1.2.9 Другие виды атак

Коротко опишем другие виды атак на алгоритмы хеширования.

Атака «встреча посередине»

Данная атака направлена на вычисление прообраза для некоего хеш-кода. Атака «встреча посередине» считается одним из вариантов атаки, использующей описанный ранее парадокс «дней рождения» [142, 202], хотя она и используется не для поиска коллизий.

Атака применима к алгоритмам хеширования, раздельно обрабатывающим различные фрагменты хешируемых сообщений, при этом возможно инвертирование данной обработки. Предположим, алгоритм хеширования hash() можно представить как совокупность алгоритма hash1(), обрабатывающего первую половину сообщения, и алгоритма hash2(), обрабатывающего вторую половину (см. рис. 1.18). Схема атаки «встреча посередине» на такой алгоритм приведена на рис. 1.19.

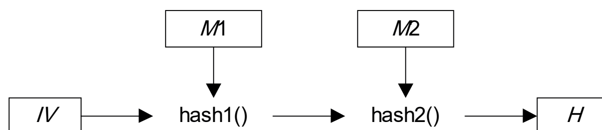

Рисунок 1.18. Пример структуры алгоритма, подверженного атаке «встреча посередине»

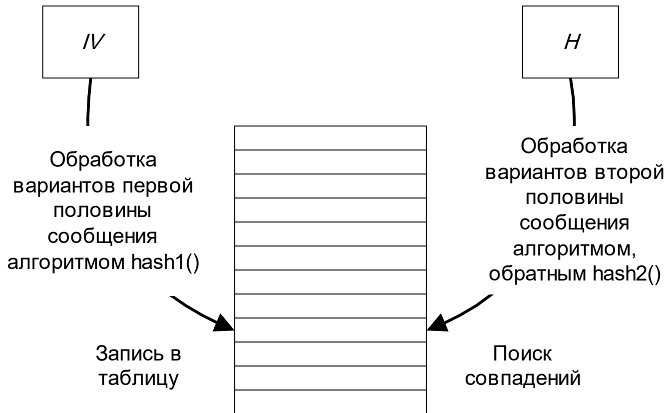

Рисунок 1.19. Атака «встреча посередине»

Цель атакующего – получение поддельного сообщения, хеш которого соответствует заданному значению *H* (предположим, некий документ с таким хеш-кодом подписан субъектом атаки). В описанной выше атаке, использующей парадокс «дней рождения», злоумышленник генерирует множество вариантов корректного и поддельного сообщений, после чего выполняет поиск коллизии. Здесь же злоумышленник генерирует множество вариантов первой половины поддельного сообщения и множество вариантов второй половины. Каждый из вариантов первой половины обрабатывается алгоритмом hash1():

$$T_x = \mathrm{hash1}(M1_x, IV),$$

где:

- $M1_x$ – один из вариантов первой половины сообщения;
- T_x – промежуточное значение;
- *IV* – начальное значение алгоритма хеширования.

Каждый из вариантов второй половины сообщения обрабатывается алгоритмом, обратным алгоритму hash2():

$$T_y = \mathrm{hash2}^{-1}(M2_y, H),$$

где $M2_y$ – один из вариантов второй половины сообщения.

Атакующий выполняет поиск совпадений значений *T*; при нахождении совпадения неких $T_i = T_j$ можно считать, что соответствующие им половины сообщений M_i и M_j формируют то самое требуемое злоумышленнику поддельное сообщение.

Таким образом, в процессе этой атаки сравниваются на совпадение не хеш-коды, а промежуточные величины. Атака существенно более эффективна, чем описанный выше поиск коллизий с помощью парадокса «дней рождения». Противодействие подобным атакам состоит в использовании в алгоритмах хеширования неинвертируемых преобразований [202].