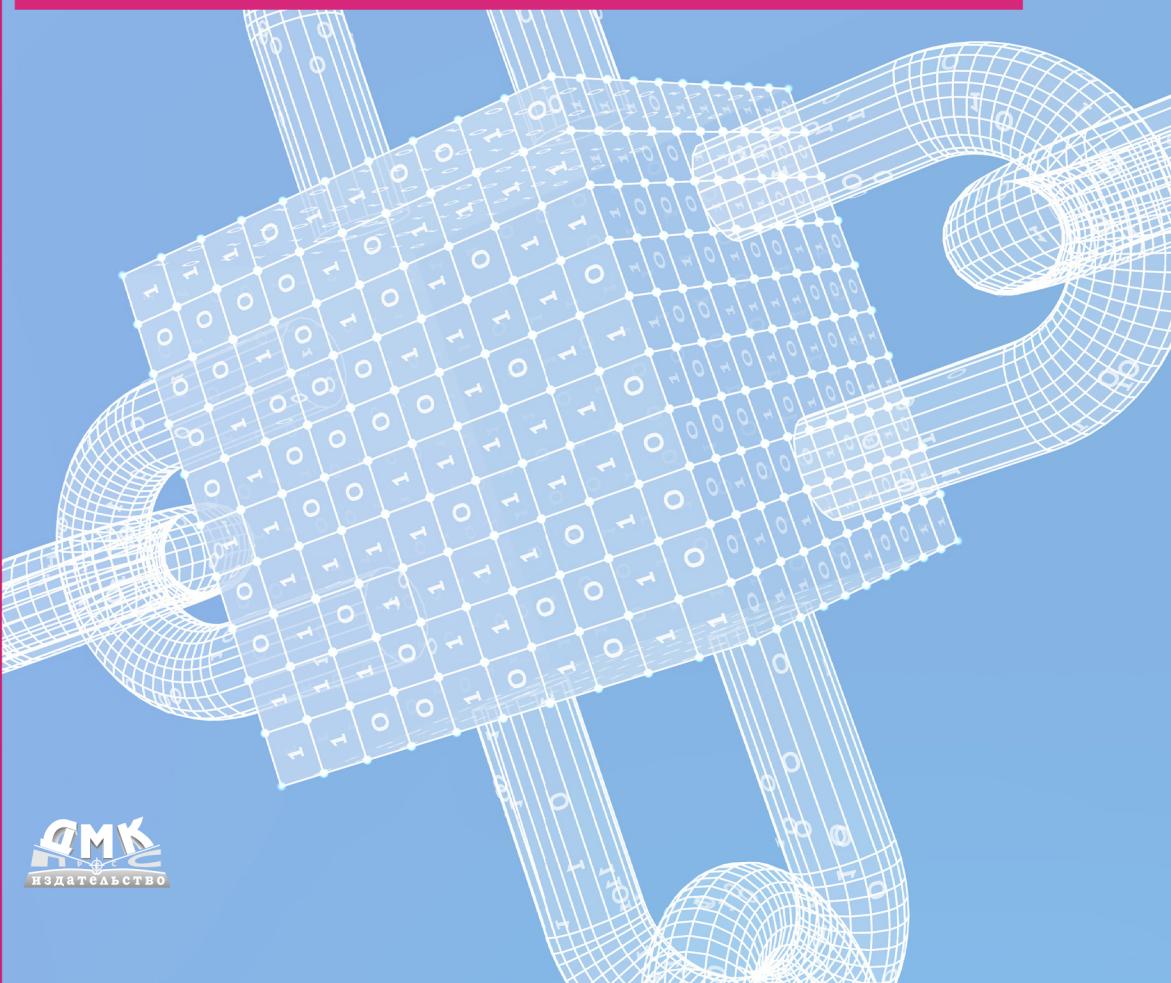


Ищукова Е. А., Панасенко С. П., Романенко К. С., Салманов В. Д.

Криптографические основы

блокчейн- технологий



Евгения Александровна Ищукова,
Сергей Петрович Панасенко,
Кирилл Сергеевич Романенко,
Вячеслав Дмитриевич Салманов

КРИПТОГРАФИЧЕСКИЕ ОСНОВЫ БЛОКЧЕЙН-ТЕХНОЛОГИЙ



Москва, 2022

УДК 004.338
ББК 65.050.253
И98

Ищукова Е. А., Панасенко С. П., Романенко К. С., Салманов В. Д.
И98 Криптографические основы блокчейн-технологий. – М.: ДМК Пресс, 2022. – 302 с.: ил.

ISBN 978-5-97060-865-4

Книга предназначена как для специалистов в области блокчейн-технологий, так и для только начинающих интересоваться данной темой. Она освещает вопросы построения блокчейн-систем, не ограничиваясь применяемыми в них криптографическими алгоритмами, но рассматривая также их основные механизмы, включая транзакции, принципы формирования блоков и сценарии достижения консенсуса в распределенных сетях. Теоретический материал книги проиллюстрирован на примере нескольких криптовалютных платформ, базирующихся на блокчейн-технологиях.

Дизайн обложки разработан с использованием ресурса [freepik.com](https://www.freepik.com)

УДК 004.338
ББК 65.050.253

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

© Ищукова Е. А., Панасенко С. П., Романенко К. С.,
Салманов В. Д., 2022

© Оформление, издание, ДМК Пресс, 2022

ISBN 978-5-97060-865-4

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 криptoанализа 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 криptoанализ	30
1.2.7 Алгебраический криptoанализ.....	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 Тараксину (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 :

$$h = f(M),$$

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 :

$$f(M_1) = f(M_2).$$

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.

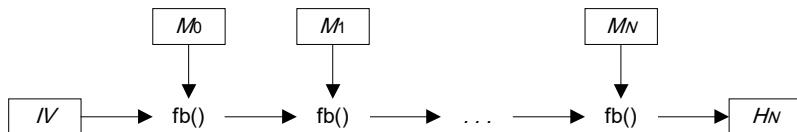


Рисунок 1.1. Схема Меркля–Дамгорда

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

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

Результатом работы алгоритма хеширования $\text{hash}()$ (т. е. хеш-кодом сообщения M) является значение H_N :

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

Начальное значение H_0 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.

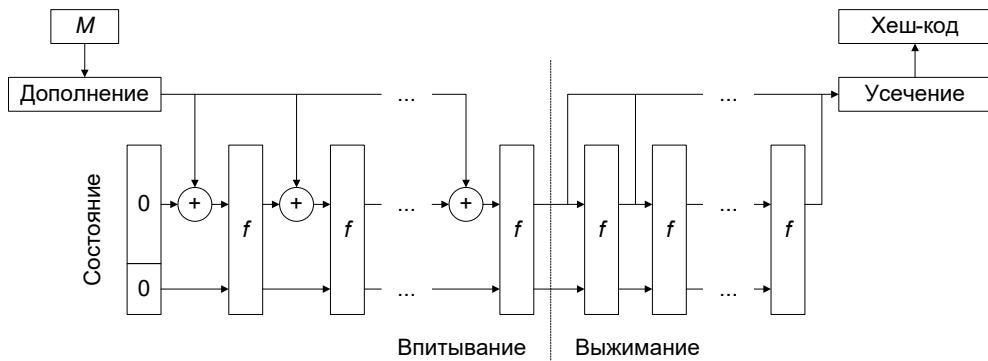


Рисунок 1.2. Фазы преобразований криптографической губки

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 односторонних 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 – ММО), 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 операция XOR сам блок M_i , в результате чего получается новое состояние алгоритма хеширования H_i .

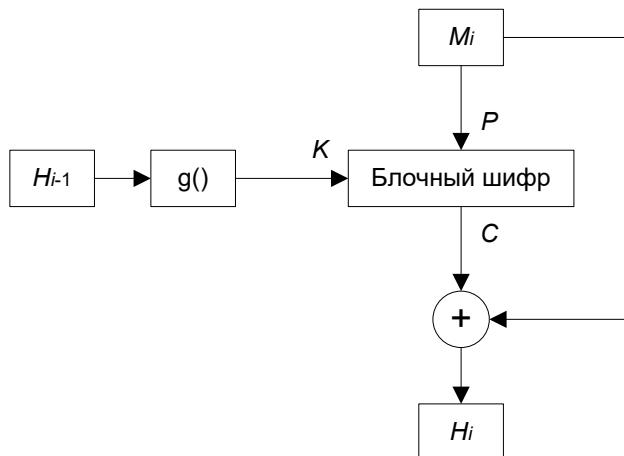


Рисунок 1.3. Схема алгоритма ММО

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 ММО 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 ММО представляется так:

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

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

Существует также вариант алгоритма ММО, 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 Миагучи–Пренеля (Miaguchi–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]:

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

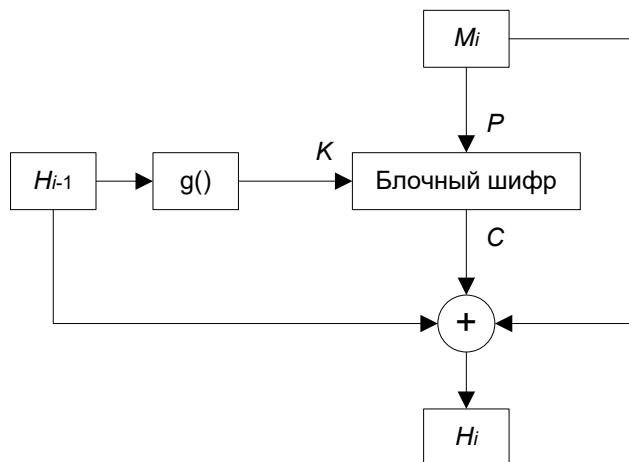


Рисунок 1.4. Схема алгоритма Миагучи–Пренеля

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.

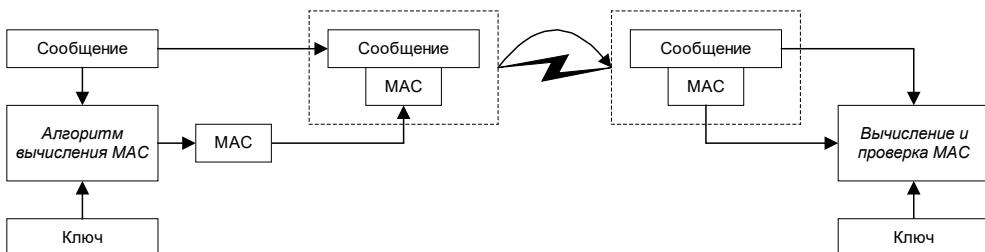


Рисунок 1.5. Схема использования MAC

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):

$$\text{MAC}(k, m) = \text{hash}(k \parallel m),$$

где:

- k – ключ;
- m – сообщение;
- $\text{hash}()$ – нижележащая бесключевая хеш-функция.

2. «Секретный суффикс» (secret suffix):

$$\text{MAC}(k, m) = \text{hash}(m \parallel k).$$

3. «Конверт» (envelope):

$$\text{MAC}(k_1, k_2, m) = \text{hash}(k_1 \parallel m \parallel k_2),$$

где подключи k_1 и k_2 являются различными.

Более безопасным способом создания МАС 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):

$$\text{key} = \text{hash}(\text{key});$$

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 $C1$, которая представляет собой блок данных, заполненный байтами с шестнадцатеричным значением 36; аналогичным образом ключ k также складывается с константой $C2$, 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 модифициированного ключа ki и сообщения m :

$$t = \text{hash}(ki \parallel m).$$

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

$$hmac = \text{hash}(ko \parallel t).$$

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:

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

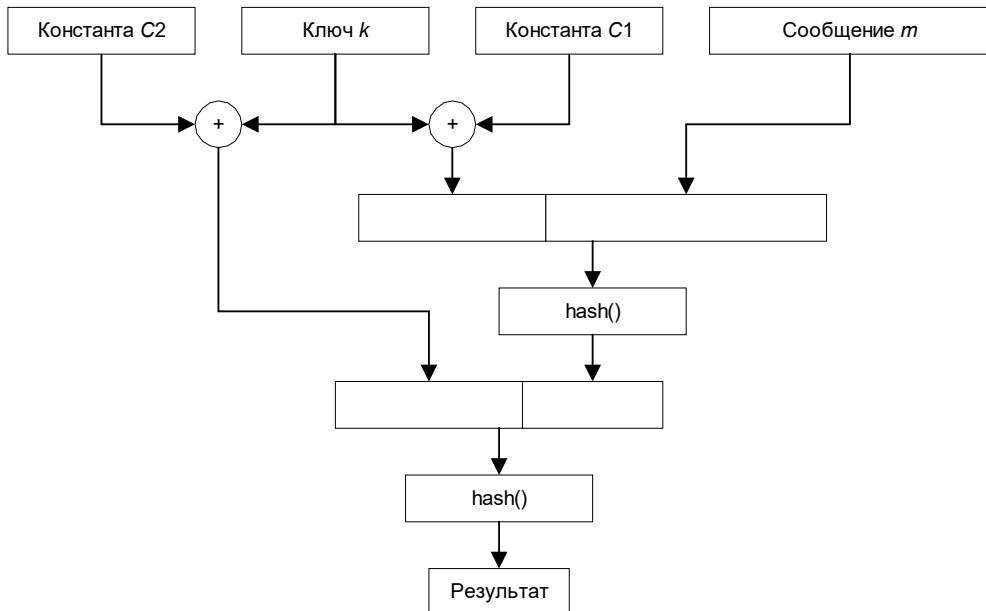


Рисунок 1.6. Вычисление HMAC

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

- «HMAC-х», где «х» – используемый алгоритм хеширования, например HMAC-MD4;
- «HMAC-х-k» в тех случаях, где выходное значение алгоритма хеширования может быть усечено (т. е. может использоваться частично); в данном случае «к» – 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:

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

где второй параметр функции $\text{hash}()$ обозначает хешируемое сообщение, а первый параметр – нестандартный вектор инициализации, в качестве которого используется ключ (соответственно, k_1 и k_2 обозначают уже выровненные до размера вектора инициализации ключи).

NMAC фактически является обобщением HMAC, поскольку ключи k_1 и k_2 можно использовать независимо, а можно представить как производные от единственного ключа k , вычисляемые следующим образом [158]:

$$k_1 = h(k \oplus C_1);$$

$$k_2 = h(k \oplus C_2),$$

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

Схема NMAC приведена на рис. 1.7.

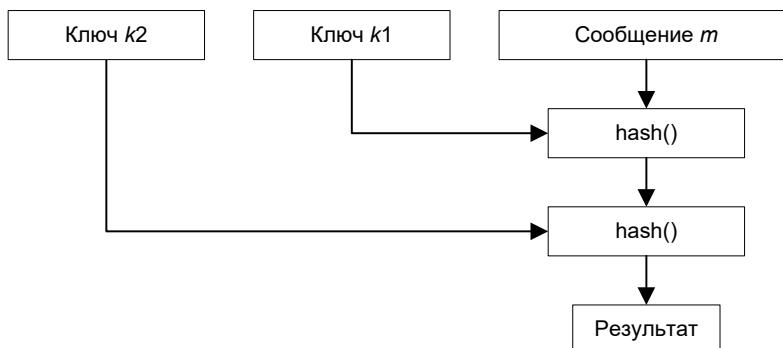


Рисунок 1.7. Вычисление NMAC

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 Мерклом в 1979 г. и запатентована в 1982 г. в патенте [178]. Общий вид дерева Меркля приведен на рис. 1.8.

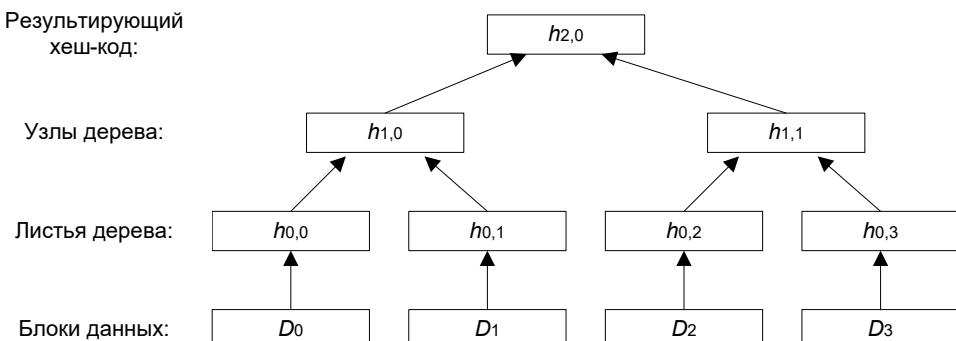


Рисунок 1.8. Дерево Меркля

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 низких уровней до вычисления корневого хеш-кода, который является результатирующим:

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

где:

- $h_{a,b}$ – b -й хеш-код a -го уровня дерева;
- \parallel – 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 криptoанализа 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 m_1 и m_2 имеют один и тот же хеш-код $h = \text{hash}(m_1) = \text{hash}(m_2)$, где $\text{hash}()$ – 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 $\text{hash}(m) = h$;
- поиск второго прообраза (second preimage attack) – при имеющемся сообщении m_1 поиск другого сообщения m_2 , удовлетворяющего условию: $\text{hash}(m_2) = \text{hash}(m_1)$.

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 $\text{hash}(IV', m) = h$, 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 $A \dots E$ – 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 $h = \text{hash}(m_1, IV_1) = \text{hash}(m_2, IV_2)$, 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. Криptoаналитик 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.

Задача 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 распаралеливается: 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 крипtosистем из-за того, что рассматривают только последовательные (т. е. использующие один атакующий компьютер) атаки на крипtosистему, не уделяя внимания более сильным параллельным атакам;
- в случае если крипtosистема 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

Смысль 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.

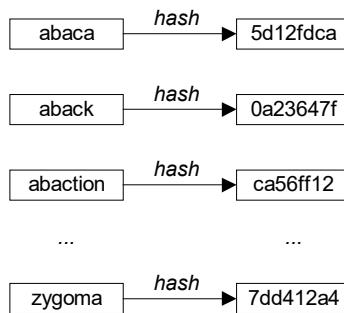


Рисунок 1.9. Таблица паролей и хеш-кодов

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:

$$V(T) = (n + k) * |P|,$$

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]:

$$p = R(h).$$

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

$$p_1 \rightarrow h_1 \rightarrow p_2 \rightarrow h_2 \rightarrow \dots \rightarrow p_N \rightarrow h_N. \quad (1.1)$$

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.

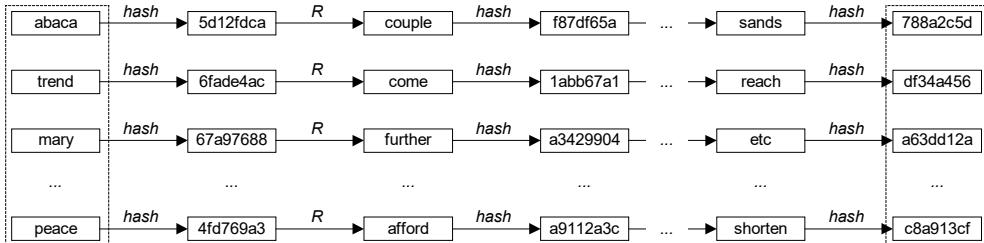


Рисунок 1.10. Таблица цепочек хеш-кодов

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

$$h_x \rightarrow p_{x+1} \rightarrow h_{x+1} \rightarrow \dots \rightarrow p_{x+N-1} \rightarrow h_{x+N-1}. \quad (1.2)$$

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.

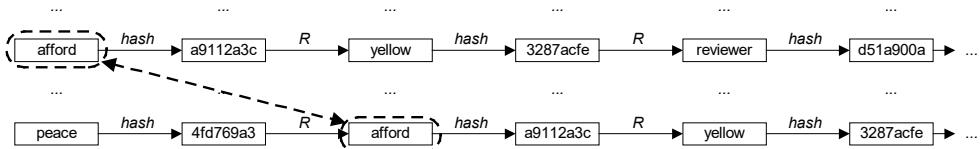


Рисунок 1.11. Пример коллизии в цепочке хеш-кодов

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 $R_1() \dots R_T()$ (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]. Стока 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».

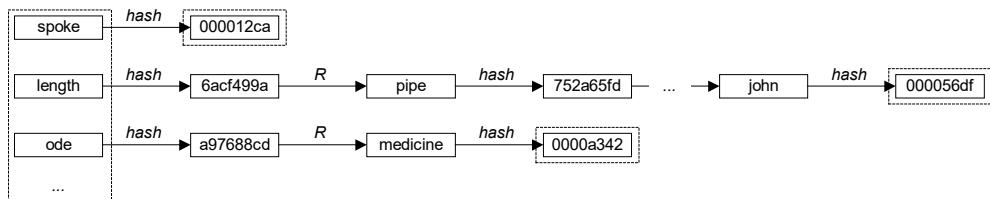


Рисунок 1.12. Таблица цепочек переменной длины

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 $R_1(\dots R_{N-1}()$ (см. рис. 1.13).

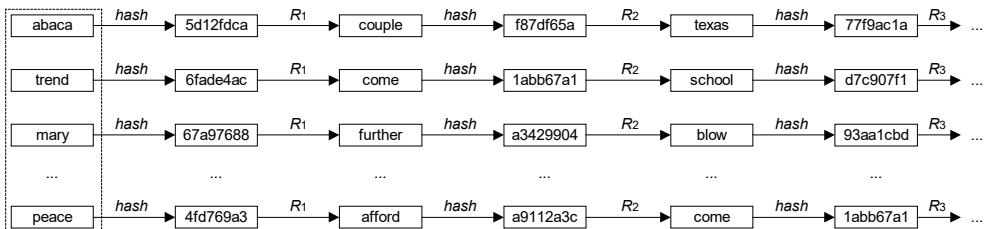


Рисунок 1.13. Радужная таблица

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 смешению 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

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

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 $R()$ 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 криptoаналитики 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:

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

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 (весыма 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.

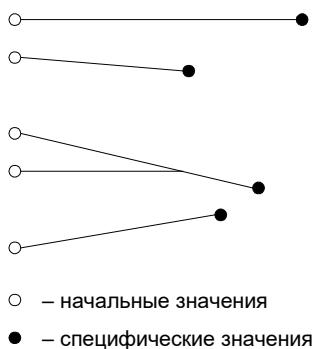


Рисунок 1.14. Параллельный поиск коллизий

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:

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

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

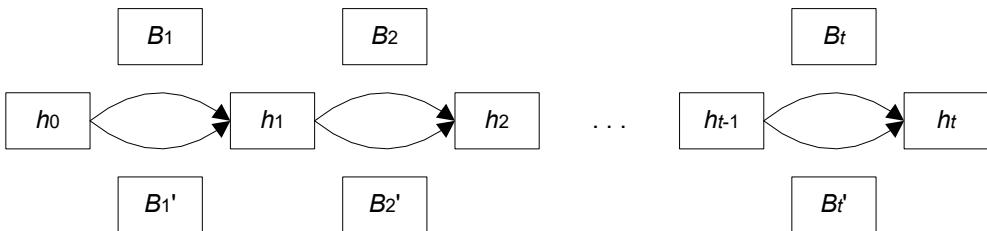


Рисунок 1.15. Мультиколлизия

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):

$$\begin{aligned}
 t &= (a \lll 5) \oplus b \oplus c \oplus d \oplus e \oplus W_i \oplus K_i; \\
 e &= d; \\
 d &= c; \\
 c &= b \lll 30; \\
 b &= a; \\
 a &= t,
 \end{aligned}$$

где:

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

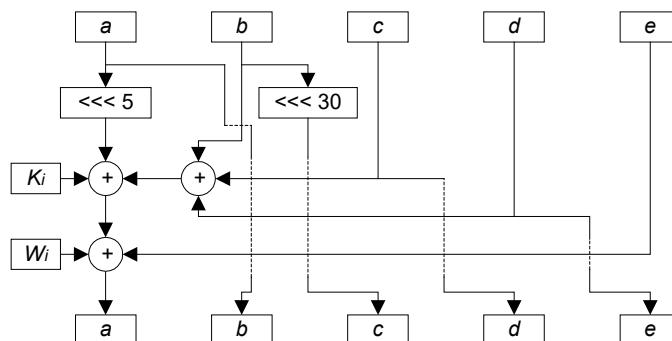


Рисунок 1.16. Итерация алгоритма SHI1

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.

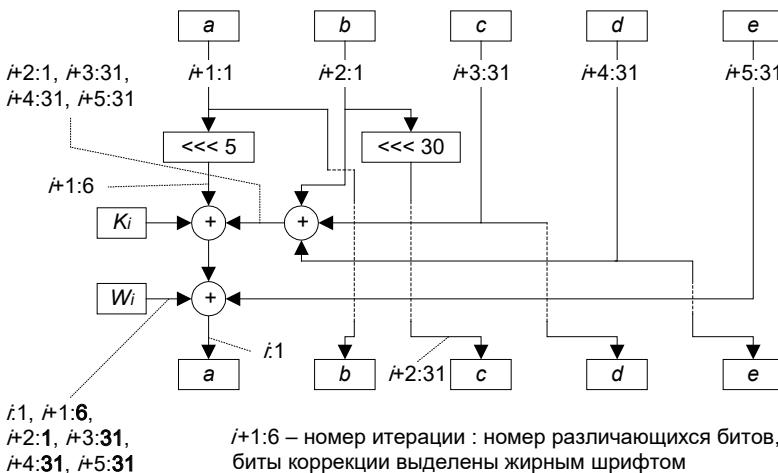


Рисунок 1.17. Распространение и коррекция разности в бите № 1 i -х фрагментов расширенных блоков алгоритма SH1

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 $a \dots e$ 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 SH1

№ 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 $W_0 \dots W_{79}$. Именно эта процедура используется для конструирования вектора возмущений (disturbance vector фактически представляет собой таблицу размером 80×32 (по числу битов в $W_0 \dots W_{79}$), 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 SH1 объясняется отсутствием в его итерации нелинейных операций.

Авторы работы [93] рассмотрели еще две упрощенные модификации алгоритма SHA, в которых в итерации SH1 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 SH1, 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 SH1 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 $\text{hash}()$ можно представить как совокупность алгоритма $\text{hash1}()$, обрабатывающего первую половину сообщения, и алгоритма $\text{hash2}()$, обрабатывающего вторую половину (см. рис. 1.18). Схема атаки «встреча посередине» на такой алгоритм приведена на рис. 1.19.

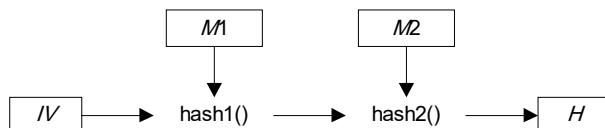


Рисунок 1.18. Пример структуры алгоритма, подверженного атаке «встреча посередине»

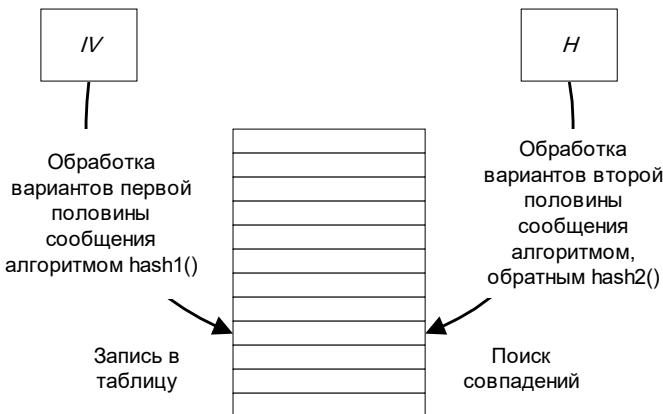


Рисунок 1.19. Атака «встречча посередине»

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 $hash1()$:

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

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 $hash2()$:

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

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].

Использование корректирующих блоков

Атака с использованием корректирующих блоков применяется в основном для поиска прообраза. Атака состоит в том, что атакующий заменяет все блоки исходного сообщения M с хеш-кодом H на какие-либо другие требующиеся ему блоки, за исключением одного или нескольких последних блоков, которые определенным образом заменяются так, чтобы хеш-код сконструированного таким образом сообщения также был равен H [202].

Эти последние блоки называются корректирующими (correcting blocks), поскольку они корректируют изменения, вносимые предыдущими блоками, формируя в результате заданный хеш-код. Наиболее часто используется один корректирующий блок.

В работе [202] показано, как данная атака может использоваться и для поиска коллизий:

- выбираются произвольные начальные сообщения M и M' ;
- эти сообщения дополняются корректирующими блоками X и X' , такими, чтобы совпадали хеш-коды результатов конкатенации начальных сообщений и корректирующих блоков $M \parallel X$ и $M' \parallel X'$.

Противодействие корректирующим блокам состоит во внесении избыточности в блоки сообщений таким образом, чтобы вычисление корректирующих блоков стало крайне ресурсоемким. Однако подобная мера ухудшает производительность систем, в которых используется хеширование [202].

Использование «неподвижных точек»

«Неподвижные точки» (fixed points) возникают, когда промежуточные вычисления хеш-кода сообщения не меняют его текущего значения.

«Неподвижная точка» возникает в том случае, когда в процессе обработки какого-либо значения M_i текущий хеш-код не изменяется, т. е. H_i остается равным H_{i-1} . В этом случае атака возможна путем добавления произвольного количества блоков со значением M_i , поскольку они не влияют на результат обработки.

Противодействие «неподвижным точкам» состоит во внедрении в хешируемые данные размера сообщения (как это сделано во многих хеш-функциях, например в хеш-функциях семейства SHA) или количества хешируемых блоков [202].

Использование манипуляций на уровне блоков

Существуют алгоритмы хеширования, позволяющие выполнять манипуляции на уровне блоков хешируемого сообщения без изменения хеш-кода, такие как удаление, вставка, замена или перестановка блоков. Многие из подобных атак пересекаются с атаками на основе корректирующих блоков, когда изменение хеш-кода из-за модификации какого-либо блока корректируется одним из следующих блоков, значение которого вычислено определенным образом для восстановления требуемого хеш-кода [202].

Специфические атаки на хеш-функции, построенные на базе блочных шифров

Как отмечено в [202], некоторые слабости блочных шифров не являются критичными в тех случаях, когда шифр используется для защиты конфиденци-

альности; однако они могут привести к успешным атакам в тех случаях, когда блочный шифр используется в качестве основы для хеш-функции.

В качестве примера таких слабостей в [202] приводятся следующие:

- свойство комплементарности ключей алгоритма шифрования DES [124];
- слабые ключи;
- неподвижные точки (имеются в виду неподвижные точки в алгоритме шифрования, когда результат зашифрования определенного сообщения на определенном ключе эквивалентен самому незашифрованному сообщению).

Эти слабости могут привести к существенному упрощению поиска коллизий (который, например, становится тривиальным при использовании свойства комплементарности ключей) или к использованию в различных целях манипуляций на уровне блоков и неподвижных точек.

Не менее критичными для использования блочных шифров в качестве основы для хеш-функций являются следующие проблемы:

- наличие эквивалентных ключей (см., например, анализ алгоритма шифрования TEA (Tiny Encryption Algorithm – «миниатюрный алгоритм шифрования») [245] в работе [157]);
- возможность атак на связанных ключах (см., например, работы [157] и [159]).

Таким образом, использование алгоритма блочного симметричного шифрования для построения хеш-функции является допустимым только в том случае, если алгоритм шифрования обладает безупречной с точки зрения криптостойкости процедурой расширения ключа.

Атаки, использующие особенности протоколов верхнего уровня

Помимо атак на алгоритмы электронной подписи, возможны атаки на различные высокоуровневые алгоритмы и протоколы, направленные на особенности использования в них алгоритмов хеширования. В частности, в [202] рекомендуются различные меры против атак повтором сообщений, а также атак, позволяющих конструировать сообщения, которые могут являться суперпозицией перехваченных злоумышленником ранее корректных сообщений. В качестве противодействия рекомендуется активное использование меток времени, случайных величин или последовательностей уникальных номеров.

1.3 НАИБОЛЕЕ ИЗВЕСТНЫЕ АЛГОРИТМЫ ХЕШИРОВАНИЯ

Опишем в данном разделе наиболее широко используемые алгоритмы хеширования.

1.3.1 Алгоритмы семейства MD

Данный раздел посвящен алгоритмам хеширования семейства MD (Message Digest). Алгоритмы MD2, MD4, MD5 и MD6 широко известны; первые три из них активно использовались в прошлом, а MD5 продолжает использоваться и сейчас, несмотря на то что существуют известные криптоаналитические атаки против данного алгоритма.

Алгоритмы MD обладают различными характеристиками; они разработаны в разные годы известным криптологом Рональдом Ривестом (Ronald Rivest)

с участием некоторых других специалистов RSA Laboratories – научного подразделения компании RSA Data Security Inc.

Алгоритм MD2

Алгоритм хеширования MD2 (который изначально назывался RSA-MD2) является первым из широко известных алгоритмов семейства MD. Существует и алгоритм MD1 (он же RSA-MD1), который, однако, является закрытым, его спецификация не опубликована. Алгоритм MD1 упоминается, в частности, в известной работе [202], а в [108] сказано, что RSA-MD1 – «закрытый алгоритм, практически идентичный алгоритму RSA-MD2».

Алгоритм MD2 был разработан в 1988 г. для использования в качестве одного из криптографических алгоритмов, входящих в стандарт защищенной электронной почты PEM (Privacy-Enhanced Mail); его реализация на языке C была приведена в RFC 1115 [167]. Впоследствии спецификация и обновленная реализация MD2 были опубликованы в RFC 1319 [155].

Структура алгоритма

MD2 хеширует данные переменного размера с получением 128-битового выходного значения. Прежде всего выполняется дополнение входных данных до размера, кратного 16 байтам (дополнение выполняется и в том случае, когда исходный размер сообщения кратен 16 байтам), следующим образом: в конец сообщения добавляется определенное количество (n) байтов (от 1 до 16); значение каждого дополнляемого байта равно n (например, дополнение 27-байтового сообщения состоит из 5 байт, каждый из которых содержит значение 5).

Затем вычисляется 16-байтовая контрольная сумма сообщения, которая также дополняет сообщение перед его дальнейшей обработкой. Контрольная сумма вычисляется следующим образом: для каждого 16-байтового блока дополненного сообщения 16 раз выполняются следующие действия:

$$\begin{aligned} c &= M[i * 16 + j]; \\ C[j] &= S[c \oplus L] \oplus C[j]; \\ L &= C[j], \end{aligned}$$

где:

- i – номер 16-байтового блока данных;
- j – номер текущего шага цикла;
- $M[x]$ – x -й байт сообщения, т. е. $M[i * 16 + j]$ – это j -й байт текущего блока данных;
- c и L – временные переменные, L изначально содержит значение 0;
- $C[j]$ – j -й байт массива контрольной суммы; перед вычислением контрольной суммы массив содержит нулевые байты;
- S – замена согласно таблице, приведенной в приложении 1. Та же таблица используется и в функции сжатия алгоритма MD2.

Функция сжатия использует 48-байтовое внутреннее состояние $X[0]…X[47]$, которое обнуляется перед началом вычислений. Каждый 16-байтовый i -й блок дополненного сообщения, включая блок контрольной суммы, накладывается на внутреннее состояние следующим образом:

1. Блок данных копируется в состояние с предварительной обработкой:

$$X[16 + j] = M[i * 16 + j], j = 0 \dots 15;$$

$$X[32 + j] = (X[16 + j] \oplus X[j]), j = 0 \dots 15.$$

Таким образом, если представить внутреннее состояние как 3 фрагмента по 16 байт, то после копирования блока данных второй из фрагментов состояния содержит обрабатываемый блок данных, а третий – результат применения побитовой операции «исключающее или» (XOR) к текущему содержимому первого фрагмента и блока данных (см. рис. 1.20).

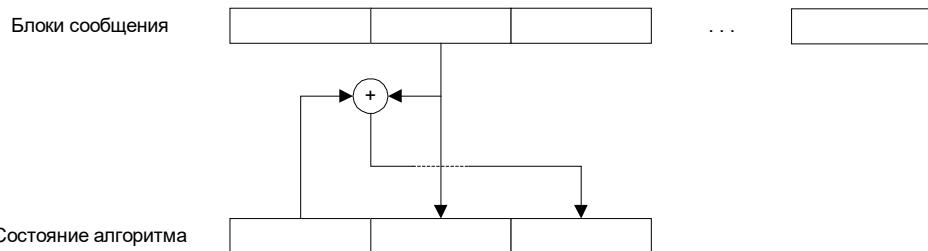


Рисунок 1.20. Копирование блока данных в состояние алгоритма MD2

- Выполняется обработка состояния, которая состоит из 18 раундов преобразований, в рамках каждого из которых выполняется обновление каждого байта состояния следующим образом:

$$X[k] = (X[k] \oplus S[t]), k = 0 \dots 47;$$

$$t = X[k],$$

где t – временная переменная, которая изначально имеет нулевое значение, а после выполнения каждого j -го раунда ($j = 0 \dots 17$) t модифицируется:

$$t = t + j \bmod 256.$$

После обработки всех блоков сообщения в первом 16-байтовом фрагменте состояния $X[0] \dots X[15]$ находится результирующий хеш-код алгоритма.

Стоит отметить, что алгоритм MD2 является крайне простым в реализации.

Быстродействие

Алгоритм MD2 является весьма медленным по сравнению с другими известными алгоритмами хеширования. Приведенные в работе [59] данные по скорости хеширования различными алгоритмами таковы:

Таблица 1.2. Быстродействие ряда известных алгоритмов хеширования

Алгоритм	Скорость хеширования, Кбит/с
MD2	78
SHA	710
RIPEMD	1334
MD5	1849
MD4	2669

Из таблицы видно, что по быстродействию алгоритм MD2 как минимум на порядок уступает другим широко используемым алгоритмам хеширования. Приведенные в таблице данные подтверждаются и другими источниками, в частности измерениями быстродействия различных реализаций алгоритмов хеширования, выполненных в рамках проекта eBASH (ECRYPT Benchmarking of All Submitted Hashes – анализ производительности хеш-функций в рамках европейского криптографического проекта ECRYPT) [111]. Отметим, что поскольку алгоритм MD2 является байт-ориентированным, на 8-битовых платформах скорость MD2 не так низка по сравнению с другими алгоритмами, как, например, на 32-битовых платформах.

Тем не менее можно сделать вывод, что MD2 является крайне медленным алгоритмом хеширования.

Результаты криptoанализа

В 1994 г. Брюс Шнайер написал про алгоритм MD2, что хотя он и более медленный, чем другие алгоритмы хеширования, но на текущий момент является криптографически стойким [53]. С тех пор криptoаналитики достигли ряда успехов в анализе MD2:

- в работе [161] предложен метод нахождения коллизии с трудоемкостью $2^{63,3}$ операций, что несколько меньше теоретической трудоемкости поиска коллизий в 2^{64} операций;
- в работе [160] описана атака на MD2, позволяющая найти прообраз с трудоемкостью от $2^{97,6}$ операций.

Следовательно, сейчас алгоритм MD2 считается взломанным.

Алгоритм MD4

Данный алгоритм интересен не только тем, что он был одним из наиболее широко применяемых алгоритмов хеширования в мире, но и тем, что на основе заложенных в MD4 идей были разработаны многие другие известные алгоритмы хеширования – прежде всего алгоритмы семейства SHA, которые будут подробно рассмотрены далее.

Как и другие алгоритмы семейства MD, MD4 разработан Рональдом Ривестом; данный алгоритм был подробно описан в 1990 г. в RFC 1186 [210].

Структура алгоритма

Структура алгоритма MD4 полностью соответствует описанной ранее схеме Меркля–Дамгорда. Алгоритм MD4 вычисляет 128-битовый хеш-код от входного блока данных переменного и неограниченного размера.

Прежде всего входное сообщение разбивается на блоки размером по 512 бит. Последний блок всегда дополняется до 512-битового размера следующим образом:

- сначала добавляется единичный бит;
- затем – нулевые биты до размера 448 бит (т. е. 512 – 64); нулевые биты не добавляются, если после добавления единичного бита блок имеет 448-битовый размер;
- к последовательности добавляется 64-битовое значение, равное размеру в битах исходной последовательности входных данных до дополнения.

В более позднем описании алгоритма [211] Ривест отметил, что если размер сообщения в битах L превышает $2^{64} - 1$ (что можно считать крайне маловероятным), то значение размера становится невозможno записать в 64 бита. Поэтому в качестве размера записывается не L , а значение $L \bmod 2^{64}$.

Если изначально размер последнего блока равен 448 бит (или больше), дополнение выполняется все равно, при этом количество блоков увеличивается на один.

Для хранения промежуточных значений алгоритм использует 4 регистра по 32 бита, которые обозначаются как A, B, C и D . Они инициализируются перед началом вычислений согласно табл. 1.3 (указаны шестнадцатеричные значения).

Таблица 1.3. Начальное заполнение регистров алгоритма MD4

Регистр	Начальное значение
A	01234567
B	89ABCDEF
C	FEDCBA98
D	76543210

Согласно описанной выше схеме Меркля–Дамгорда, совокупность регистров $A \dots D$ содержит значение H_i после обработки i -го блока сообщения, а указанное в таблице начальное заполнение регистров является вектором инициализации алгоритма.

Каждый из 512-битовых блоков входных данных M_i поочередно обрабатывается следующим образом:

- Блок входных данных представляется в виде 16 слов $M[0] \dots M[15]$, каждое из которых является 32-битовым.
- Содержимое регистров $A \dots D$ копируется во временные переменные $a \dots d$.
- Выполняется 48 итераций преобразований (см. рис. 1.21), в каждой из которых модифицируются временные переменные $a \dots d$ с участием одного из слов блока сообщения $M[0] \dots M[15]$:

$$\begin{aligned} t = & (a + f_j(b, c, d) + M[x_j] + K_j \bmod 2^{32}) \lll s; \\ a = & d; \\ d = & c; \\ c = & b; \\ b = & t, \end{aligned}$$

где:

- j – номер итерации, $j = 0 \dots 47$;
- t – временная 32-битовая переменная;
- $f_j()$ – одна из следующих раундовых функций:

Таблица 1.4. Раундовые функции алгоритма MD4

Номер итерации j	$f_j()$
0...15	$f_j(x, y, z) = (x \& y) (\sim x \& z)$
16...31	$f_j(x, y, z) = (x \& y) (x \& z) (y \& z)$
32...47	$f_j(x, y, z) = x \oplus y \oplus z$

Здесь и далее символами $\&$ и $|$ обозначены, соответственно, побитовые логические операции «и» и «или»; $\sim x$ обозначает побитовый комплемент к x ;

- x_j – индекс используемого слова блока сообщения; индексы x_j приведены в приложении 1;
- K_j – модифицирующие константы (указаны шестнадцатеричные значения):

Таблица 1.5. Модифицирующие константы алгоритма MD4

Номер итерации j	K_j
0...15	0
16...31	5A827999
32...47	6ED9EBA1

- $<<<$ – операция побитового циклического сдвига влево; количество битов сдвига s_j определяется в зависимости от номера итерации согласно табл. 1.6.

Таблица 1.6. Количество битов сдвига в зависимости от номера итерации алгоритма MD4

Номер итерации j	s_j
0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44	3
17, 21, 25, 29	5
1, 5, 9, 13	7
18, 22, 26, 30, 33, 37, 41, 45	9
2, 6, 10, 14, 34, 38, 42, 46	11
19, 23, 27, 31	13
35, 39, 43, 47	15
3, 7, 11, 15	19

Последняя итерация при обработке каждого блока сообщения отличается от остальных тем, что в ней выполняется только следующая операция:

$$a = (a + f_j(b, c, d) + M[x_j] + K_j \bmod 2^{32}) <<< s_j.$$

4. Значения регистров $A \dots D$ складываются по модулю 2^{32} с полученными значениями переменных $a \dots d$ соответственно.

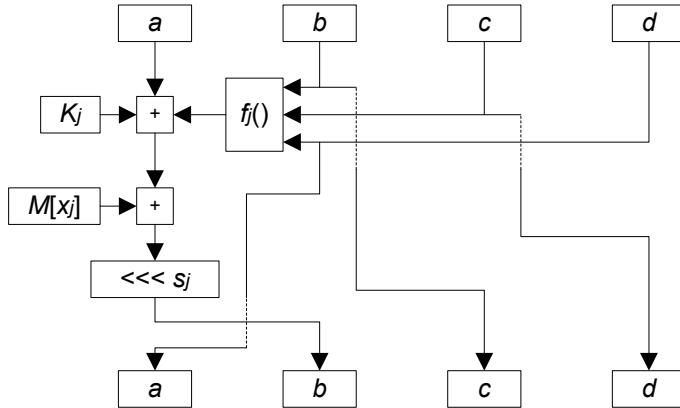


Рисунок 1.21. Итерация алгоритма MD4

Выходное значение алгоритма, т. е. 128-битовый хеш-код сообщения, – результат конкатенации регистров $A \dots D$ после обработки последнего блока расширенного сообщения.

Алгоритм MD4 выглядит не более сложным в реализации, чем описанный в предыдущем разделе алгоритм MD2. В отличие от MD2, MD4 не содержит таблиц замен и может быть весьма компактно реализован аппаратно [211].

Расширенный вариант алгоритма

В [210] описан также расширенный вариант алгоритма MD4, который позволяет вычислить 256-битовый хеш-код. Расширенный вариант основан на параллельном применении двух копий обычного алгоритма MD4. Вычисления выполняются следующим образом:

1. Первая копия алгоритма выполняется обычным образом, за исключением описанного далее обмена значениями регистров A .
2. Во второй копии применяются другие начальные значения регистров $A \dots D$ согласно таблице (в этой и следующей таблицах указаны шестнадцатеричные значения регистров и констант):

Таблица 1.7. Начальное заполнение регистров в расширенном варианте алгоритма MD4

Регистр	Начальное значение
A	00112233
B	44556677
C	8899AABB
D	CCDDEEFF

3. В итерациях 16...47 используются другие значения модифицирующих констант K_j :

Таблица 1.8. Модифицирующие константы в расширенном варианте алгоритма MD4

Номер итерации j	K_j
16...31	50A28BE6
32...47	5C4DD124

4. После обработки каждого из 512-битовых блоков входных данных (в т. ч. после обработки последнего блока) значения регистров A двух копий алгоритма MD4 меняются местами.

Результирующим 256-битовым хеш-кодом алгоритма является конкатенация регистров $A...D$ первой копии MD4 и регистров $A...D$ второй копии после обработки всех блоков входных данных. Отметим, что расширенный вариант алгоритма MD4 продолжает соответствовать схеме Меркля–Дамгорда.

Из более поздней спецификации алгоритма MD4 ([211]) расширенный вариант исключен.

Результаты криптоанализа

Криптоанализ алгоритма MD4 во многом шел параллельно с криптоанализом более поздних алгоритмов, частично унаследовавших от MD4 его структуру и используемые в нем преобразования. Прежде всего это алгоритм MD5, а также бывшие стандарты хеширования США SHA и SHA-1; данные алгоритмы описаны ниже.

Алгоритм MD4 активно использовался до начала 2000-х гг. – до тех пор, когда была предложена эффективная атака на этот алгоритм, позволяющая практически мгновенно найти коллизию.

На данный момент алгоритм считается полностью взломанным (статус «исторический» был присвоен алгоритму MD4 в 2011 г. [237]) по причине следующих результатов его криптоанализа:

- мгновенное (достаточно двух операций хеширования) нахождение коллизии [219];
- описанная в работе [166] атака, позволяющая найти прообраз с трудоемкостью 2^{102} операций.

Алгоритм MD5

Алгоритм MD5 был предложен в 1992 г., т. е. всего через два года после опубликования алгоритма MD4. Предпосылки разработки MD5 были таковы [212]:

- при разработке алгоритма MD4 основным критерием было быстродействие данного алгоритма, его криптостойкость считалась тогда менее важной; в результате уже в 1991 г. была опубликована первая атака на усеченный алгоритм MD4 [104];
- поскольку алгоритм MD4 получился исключительно быстрым, особенно на 32-битовых платформах, предполагалось, что именно поэтому он будет в центре внимания криптоаналитиков, что рано или поздно приведет к обнаружению уязвимостей и к успешным криптоаналитическим атакам на данный алгоритм.

В результате Рональд Ривест разработал алгоритм MD5 – несколько более медленный, чем MD4, но обладающий более консервативным дизайном и учитывающий текущие результаты и тенденции в криптоанализе, что предопределило несколько более высокую, по сравнению с MD4, криптостойкость данного алгоритма.

Структура алгоритма

Как и алгоритм MD4, MD5 вырабатывает 128-битовый хеш-код для сообщений переменной длины. MD5 подробно описан в документе [212].

Прежде всего, аналогично алгоритму MD4, входное сообщение разбивается на блоки по 512 бит; последний из блоков всегда дополняется до 512-битового размера следующим образом:

- добавляется единичный бит;
- затем – нулевые биты до размера 448 бит; нулевые биты не добавляются, если после добавления единичного бита блок уже имеет 448-битовый размер;
- к последовательности добавляется 64-битовое значение, равное размеру в битах исходной последовательности входных данных до дополнения; если размер сообщения превышает $2^{64} - 1$, то записывается значение размера по модулю 2^{64} .

Аналогично алгоритму MD4, для хранения промежуточных значений после обработки каждого блока сообщения используются 32-битовые регистры A , B , C и D . Вектор инициализации алгоритма MD5 эквивалентен таковому в MD4 (указаны шестнадцатеричные значения):

Таблица 1.9. Начальное заполнение регистров алгоритма MD5

Регистр	Исходное значение
A	01234567
B	89ABCDEF
C	FEDCBA98
D	76543210

512-битовые блоки дополненного сообщения M_i поочередно обрабатываются следующим образом:

1. Блок входных данных представляется в виде 16 слов $M[0]...M[15]$, каждое из которых является 32-битовым.
2. Содержимое регистров $A...D$ копируется во временные переменные $a...d$.
3. Выполняются 64 итерации преобразований (см. рис. 1.22), в каждой из которых модифицируются переменные $a...d$ с участием одного из слов блока сообщения $M[0]...M[15]$:

$$\begin{aligned} t &= b + ((a + f_j(b, c, d) + M[x_j] + T_j \bmod 2^{32}) \ll s_j) \bmod 2^{32}; \\ a &= d; \\ d &= c; \end{aligned}$$

$$c = b;$$

$$b = t,$$

где:

- j – номер итерации, $j = 1 \dots 64$ (каждые 16 итераций формируют один из четырех раундов алгоритма);
- t – временная 32-битовая переменная;
- $f_j()$ – одна из следующих раундовых функций:

Таблица 1.10. Раундовые функции алгоритма MD5

Номер итерации j	$f_j()$
1...16	$f_j(x, y, z) = (x \& y) \mid (\sim x \& z)$
17...32	$f_j(x, y, z) = (x \& z) \mid (y \& \sim z)$
33...48	$f_j(x, y, z) = x \oplus y \oplus z$
49...64	$f_j(x, y, z) = y \oplus (x \mid \sim z)$

- x_j – индексы используемых слов блока сообщения; приведены в приложении 1;
- T_j – модифицирующие константы; приведены в приложении 1;
- \lll – операция побитового циклического сдвига влево; количество битов сдвига s_j определяется в зависимости от номера итерации согласно таблице, приведенной в приложении 1.

Последняя итерация при обработке каждого блока сообщения отличается от остальных тем, что в ней выполняется только следующая операция:

$$a = b + ((a + f_j(b, c, d) + M[x_j] + T_j \bmod 2^{32}) \lll s_j) \bmod 2^{32}.$$

4. Значения регистров $A \dots D$ складываются по модулю 2^{32} с полученными значениями переменных $a \dots d$ соответственно.

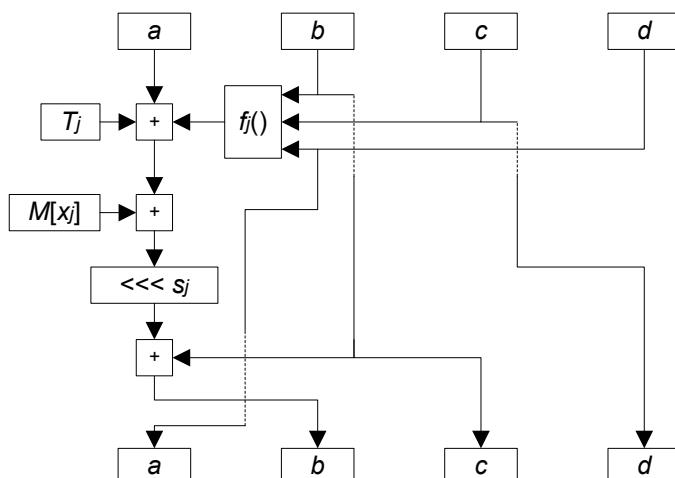


Рисунок 1.22. Итерация алгоритма MD5

Выходное значение алгоритма, т. е. 128-битовый хеш-код сообщения, – результат конкатенации регистров $A \dots D$ после обработки последнего блока расширенного сообщения.

Основные отличия от MD4

Основные отличия алгоритма MD5 от MD4 таковы [212]:

- в алгоритме выполняется 4 раунда вместо трех;
- изменены раундовые функции $f_i()$, в частности больше не используется функция $f_j(x, y, z) = (x \& y) | (x \& z) | (y \& z)$, свойства которой эксплуатировались в ряде атак на алгоритм MD4 (см., например, [241]);
- в каждую итерацию добавлена дополнительная операция сложения с переменной b (отметим, что именно переменная b модифицируется в каждой предыдущей итерации, за исключением первой) по модулю 2^{32} , ускоряющая влияние значений битов входных данных на биты выходного значения;
- в каждом раунде теперь используются уникальные модифицирующие константы;
- изменен (на менее упорядоченный) порядок обработки входных слов блока сообщения, а также оптимизированы величины s_j .

Алгоритм MD5 не выглядит более сложным в реализации, чем алгоритм MD4. Все изменения в алгоритме являются достаточно локальными, но в совокупности автору алгоритма удалось решить задачу его усиления за счет некоторого ухудшения скоростных качеств алгоритма (согласно [174], MD4 примерно в 1,5 раза быстрее, чем MD5).

Результаты криптоанализа

За время активного использования алгоритма MD5 было предпринято множество попыток атаковать данный алгоритм. История криптоанализа MD5 весьма многогранна. Она тесно переплетается с криптоанализом ряда других алгоритмов хеширования (в том числе рассмотренного ранее MD4); можно утверждать, что многие достижения в криптоанализе алгоритмов хеширования и успешные атаки на их применения были сделаны благодаря алгоритму MD5 и его популярности, весьма привлекавшей криптоаналитиков со всего мира.

Достижения в криптоанализе алгоритма MD5 привели также к широкому спектру атак на приложения и протоколы, использующие данный алгоритм; многие из этих атак привели к значительным прорывам в криптоанализе (см. обзор в [35]).

На текущий момент достигнуты значительные результаты в криптоанализе алгоритма MD5:

- нахождение коллизий с достаточно незначительной трудоемкостью от 2^{16} операций [230];
- нахождение прообразов с трудоемкостью $2^{123,4}$ операций [170, 218].

Алгоритм MD5 все еще довольно широко используется в применениях, где не требуется стойкость к нахождению коллизий, несмотря на множество известных атак на подобные приложения и рекомендации экспертов не использовать данный алгоритм [246].

Алгоритм MD6

Следующий после алгоритма MD5 и наиболее современный из алгоритмов хеширования данного семейства – это алгоритм MD6.

MD6 был разработан большой группой криптологов из Массачусетского технологического института (Massachusetts Institute of Technology – MIT) в 2008 г. под руководством Рональда Ривеста. Данный алгоритм был представлен на конференции Crypto 2008 [213], а также был предложен на конкурс по определению нового стандарта США SHA-3 (описан далее).

MD6 – алгоритм, имеющий большое количество опциональных параметров, с помощью которых он настраивается под разнообразные возможные реализации и применения. Рассмотрим сначала базовый вариант алгоритма, использующий параметры по умолчанию, а затем опишем его альтернативные варианты.

Входные и выходные данные алгоритма

Согласно спецификации [214], алгоритм MD6 обрабатывает любое сообщение M , размер которого в битах m соответствует следующим ограничениям:

$$0 \leq m < 2^{64}.$$

Размер сообщения не обязательно должен быть известен заранее (т. е. до начала хеширования), что позволяет обрабатывать потоковые данные по мере их поступления.

Вырабатываемые алгоритмом MD6 хеш-коды могут быть любого положительного размера d , не превышающего 512 бит. Размер выходного значения алгоритма должен быть известен заранее, поскольку значение d участвует в вычислениях на различных этапах с самого начала обработки входного сообщения.

Варианты алгоритма, использующие значения опциональных параметров по умолчанию, обозначаются как MD6- d . Основными вариантами алгоритма можно считать варианты с размерами хеш-кодов, обязательными для участия алгоритма в конкурсе SHA-3 (см., например, [145]), а именно:

- MD6-224;
- MD6-256;
- MD6-384;
- MD6-512.

Кроме того, в качестве одного из основных авторы алгоритма MD6 рассматривают вариант MD6-160, поскольку он совместим по размеру со стандартом SHA-1 [126].

Структура алгоритма

Структура алгоритма MD6 принципиально отличается от рассмотренных ранее алгоритмов семейства MD. Базовый вариант алгоритма обрабатывает входные данные, структурированные в виде кватернарного дерева.

Данное дерево представлено на рис. 1.23. Листьями дерева являются блоки хешируемого сообщения M и блоки дополнения, которые обозначены на рис. 1.23 круглыми элементами с различной окраской:

- черные соответствуют блокам сообщения;
- серый обозначает последний неполный блок сообщения (если таковой существует), частично дополненный до размера блока;
- белые – блоки дополнения.

Алгоритм оперирует блоками размером в 16 слов; словом является 64-битовый фрагмент данных, т. е. размер блока составляет 128 байт, или 1024 бита.

Какой-либо вектор инициализации в алгоритме не предусмотрен. Дополнение хешируемого сообщения также выполняется максимально просто – добавлением нулевых битов до требуемого размера.

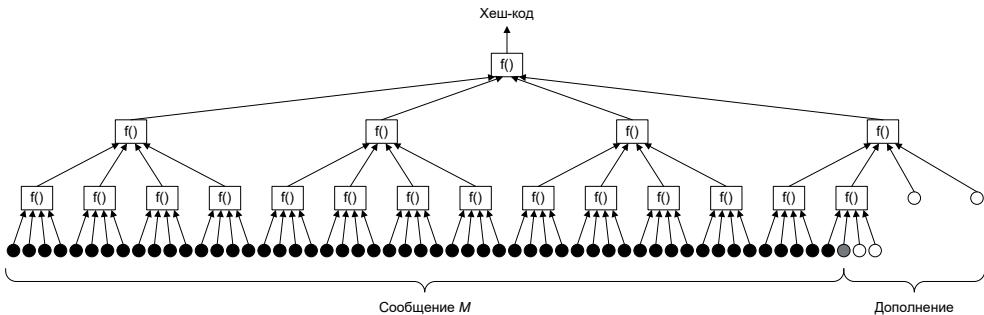


Рисунок 1.23. Структура алгоритма MD6

Каждый узел дерева соответствует применению функции сжатия $f()$. Функция сжатия получает в качестве входного значения 4 блока данных, которые сжимаются до одного блока, передаваемого «вверх» следующей функции сжатия, как показано на рис. 1.23. Последнее применение функции сжатия дает 1024-битовое значение, которое усекается до требуемого размера d , в результате чего получается хеш-код сообщения M . Функция сжатия будет подробно описана далее.

Данная древовидная структура алгоритма выбрана для достижения максимальной возможности по распараллеливанию хеширования между несколькими процессорными ядрами. Действительно, все экземпляры функции $f()$, находящиеся на одном уровне дерева, могут быть выполнены параллельно, поскольку они полностью независимы друг от друга.

Отметим, что существуют альтернативные структуры алгоритма, которые не применяются в его базовом варианте; данные структуры, а также обоснование необходимости наличия альтернативных структур алгоритма будут обсуждаться далее.

Входная последовательность функции сжатия

Помимо четырех блоков входных данных по 16 слов, функция сжатия получает еще несколько входных параметров, которые вместе с данными формируют 89 слов входной последовательности, обрабатываемой функцией сжатия.

Структура входной последовательности приведена на рис. 1.24, где размеры фрагментов последовательности указаны в 64-битовых словах и использованы следующие обозначения:

- Q – константа, зависящая от порядкового номера слова входной последовательности; значения констант приведены в приложении 1;

- K – значение секретного ключа; в базовом варианте алгоритма секретный ключ не используется, поэтому данное поле обнулено;
- U – уникальный идентификатор конкретного экземпляра функции сжатия, определяется местоположением узла, соответствующего данному экземпляру, в древовидной структуре алгоритма;
- V – дополнительное слово, содержащее значения остальных параметров функции сжатия.

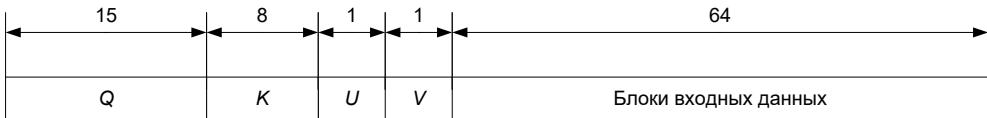


Рисунок 1.24. Входная последовательность функции сжатия алгоритма MD6

Рассмотрим фрагменты входной последовательности более подробно.

Структура слова U приведена на рис. 1.25, где размеры фрагментов приведены в байтах. Значение U формируется следующими переменными:

- $level$ – номер уровня узла дерева; узлы нумеруются снизу вверх, начиная со значения 0, которое соответствует уровню блоков сообщения M ;
- $index$ – позиция узла в текущем уровне дерева, считается слева направо, начиная с 0.



Рисунок 1.25. Идентификатор экземпляра функции сжатия U в алгоритме MD6

Дополнительное слово V имеет значительно более сложную структуру, но в базовом варианте многие из составляющих V параметров имеют константные значения. Структура V приведена на рис. 1.26, где размеры фрагментов приведены в битах, а параметры обозначены следующим образом:

- RFU – зарезервированное поле (Reserved for Future Usage), заполненное нулями;
- r – количество раундов функции сжатия (см. далее); в базовом варианте напрямую зависит от d (размер хеш-кода в битах, также представленный в слове V) следующим образом:

$$r = 40 + \lfloor d / 4 \rfloor;$$

- L – параметр, определяющий структуру алгоритма; в базовом варианте $L = 64$;
- z – параметр, определяющий, является ли данное вычисление функции сжатия финальным (т. е. корнем дерева на рис. 1.23); в этом случае $z = 1$, во всех остальных узлах $z = 0$;
- p – количество битов дополнения (если таковые существуют) в блоках входных данных;

- *keylen* – размер ключа K в байтах; поскольку в базовой версии ключ K не используется, данное поле обнулено.



Рисунок 1.26. Дополнительное слово V входной последовательности функции сжатия алгоритма MD6

Функция сжатия

Функция сжатия выполняет t итераций обработки входной последовательности; количество итераций определяется количеством раундов r : $t = 16r$.

Обозначим как N входную последовательность (размером в 89 слов), являющуюся результатом конкатенации Q, K, U, V и блоков входных данных, как показано на рис. 1.24. Обозначим также i -е слово любой последовательности X как X_i .

Перед выполнением первой итерации функции сжатия выполняется инициализация ее внутренних данных следующим образом:

1. Создается массив A размером $t + 89$ слов: $A_0 \dots A_{t+88}$.
2. Входная последовательность N копируется в первые 89 слов массива A :

$$A_i = N_i, i = 0 \dots 88.$$

Каждая итерация модифицирует одно слово массива A (а именно A_i для $i = 89 \dots t + 88$) следующим образом (см. рис. 1.27):

$$\begin{aligned} x &= S_{i-89} \oplus A_{i-89} \oplus A_{i-t0}; \\ x &= x \oplus (A_{i-t1} \& A_{i-t2}) \oplus (A_{i-t3} \& A_{i-t4}); \\ x &= x \oplus (x \gg r_{i-89}); \\ A_i &= x \oplus (x \ll l_{i-89}), \end{aligned}$$

где:

- x – временная переменная;
- S_j – раундовые константы, будут описаны далее;
- $t0 \dots t4$ – константы, определяющие позиции обратной связи; в базовом варианте алгоритма определены следующим образом:

Таблица 1.11. Позиции обратной связи в итерации алгоритма MD6

$t0$	$t1$	$t2$	$t3$	$t4$
17	18	21	31	67

- r_j и l_j – константы, определяющие количество битов вращения текущего значения x , соответственно, вправо и влево; значения констант приведены в приложении 1.

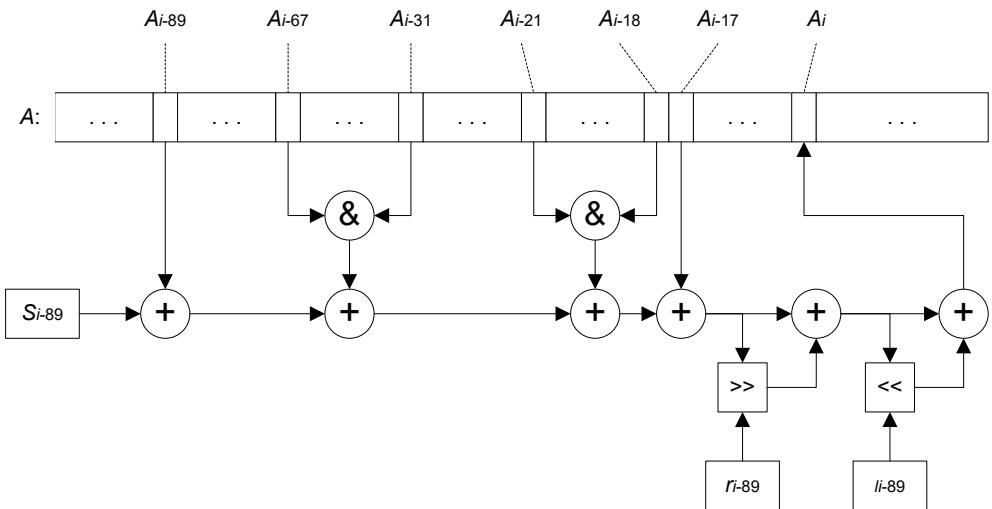


Рисунок 1.27. Итерация функции сжатия алгоритма MD6

Раундовые константы S_j определены на основе двух следующих констант (приведены в шестнадцатеричном виде):

$$S'_0 = 0123456789abcdef;$$

$$S^* = 7311c2812425cf0a.$$

Вычисление раундовых констант выполняется так:

$$S'_{i-89} = S'_{\lfloor (i-89)/16 \rfloor},$$

где $S'_{k+1} = (S'_k \lll 1) \oplus (S'_k \& S^*)$.

На рис. 1.27 отчетливо видно, что в данном случае имеет место сдвиговый регистр с нелинейной обратной связью, на основе которого и может быть реализована функция сжатия алгоритма MD6.

Обработка всего массива \$A\$ на основе его первых 89 слов в [214] очень наглядно представлена в виде сдвигающегося «окна», поочередно захватывающего для обработки 89 элементов массива и формирующего за одну итерацию значение одного следующего элемента. Данное представление функции сжатия приведено на рис. 1.28.

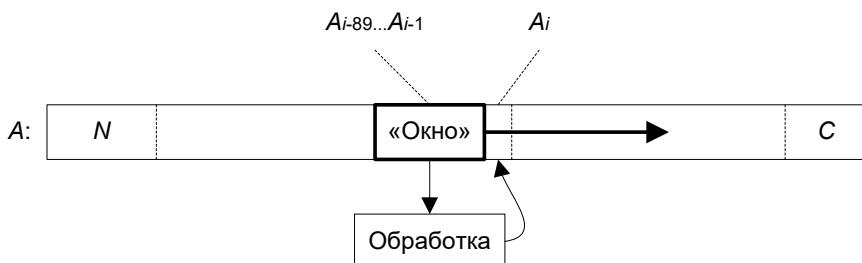


Рисунок 1.28. Функция сжатия алгоритма MD6

Результат работы функции сжатия обозначен на рис. 1.28 символом C . Это массив размером в 16 слов, которые представляют собой последние 16 слов массива A , т. е.:

$$C_i = A_{t+73+i}, i = 0 \dots 15.$$

Резюмируя описание базового варианта алгоритма MD6, можно сказать следующее:

- алгоритм имеет оригинальную древовидную структуру; листьями дерева являются блоки хешируемого сообщения и дополнения, а узлы соответствуют применению функции сжатия алгоритма;
- алгоритм фактически не имеет внутреннего состояния, а результат работы конкретной функции сжатия передается на вход функции сжатия родительского узла;
- алгоритм имеет множество дополнительных параметров, которые мы обсудим далее.

Альтернативные варианты структуры

Один из важнейших параметров алгоритма MD6 – это параметр L (был упомянут ранее, поскольку входит в состав дополнительного слова V – комплексного параметра функции сжатия алгоритма). Данный параметр определяет «степень параллельности» алгоритма MD6 и непосредственно влияет на структуру алгоритма.

Параметр L может принимать любое целочисленное значение в диапазоне от 0 до 64 включительно. По умолчанию $L = 64$, в этом случае алгоритм имеет структуру в виде квaternionного дерева, которая была приведена на рис. 1.23. Данная структура предоставляет максимальные возможности по распараллеливанию вычислений.

Противоположный вариант структуры при $L = 0$ приведен на рис. 1.29. Этот вариант предполагает строго последовательное вычисление хеш-кодов путем поочередной обработки блоков входных данных.

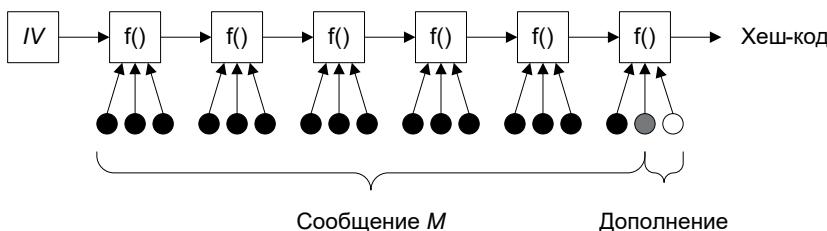


Рисунок 1.29. Последовательный вариант алгоритма MD6

В данной структуре возможности по распараллеливанию вычислений существуют только в рамках выполнения функции сжатия $f()$, на верхнем уровне вычисления выполняются строго последовательно. С другой стороны, такая структура предъявляет значительно меньшие требования к оперативной памяти и поэтому может быть с успехом применена в устройствах с ограниченными ресурсами.

При $L = 0$ оригинальная древовидная структура алгоритма полностью вырождается и становится аналогичной схеме Меркля–Дамгорда (в варианте

с удвоенным размером внутреннего состояния алгоритма относительно размера выходного значения – Double-pipe Merkle-Damgård Construction), которая была рассмотрена нами ранее.

В такой схеме алгоритм MD6 имеет обычные атрибуты схемы Меркля–Дамгårда:

- вектор инициализации IV – заполненный нулями блок из 16 слов;
- состояние алгоритма (начальным значением которого является IV) – результат выполнения функции сжатия на промежуточных этапах.

Выходное значение алгоритма образуется усечением до d бит результата выполнения последней функции сжатия.

Поскольку, в отличие от древовидной структуры, необходимо каким-то образом передавать текущее состояние, функция сжатия обрабатывает не по 4, а по 3 блока (каждый блок состоит из 16 слов) данных; вместо одного из блоков данных функция сжатия принимает на вход текущее состояние алгоритма. Таким образом, входная последовательность функции сжатия отличается по своей структуре от входной последовательности функции сжатия базового варианта алгоритма и имеет структуру, изображенную на рис. 1.30, где текущее состояние алгоритма обозначено как S .

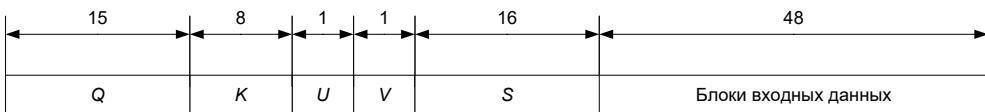
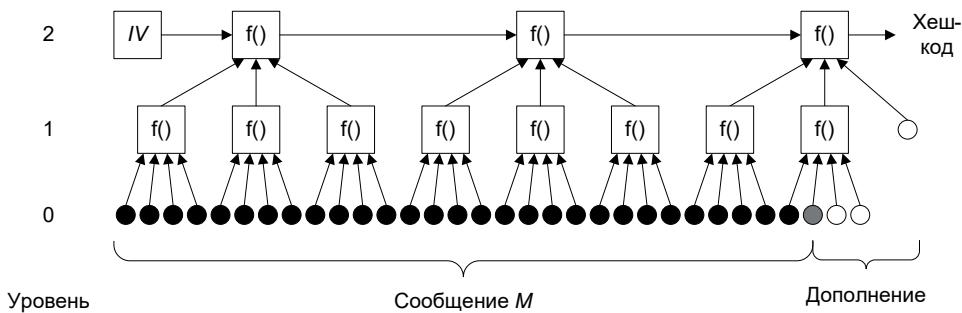


Рисунок 1.30. Входные данные функции сжатия при последовательном вычислении хеш-кода алгоритмом MD6

Допускаются промежуточные значения L между 0 и 64 – в общем случае L определяет максимально допустимое количество уровней в древовидной структуре алгоритма. Уровни нумеруются снизу вверх, начиная с нуля, при этом нулевой уровень всегда обозначает уровень блоков хешируемого сообщения. Например, в структуре на рис. 1.23 содержатся уровни с нулевого по третий включительно, а структура, приведенная на рис. 1.29, состоит из нулевого и первого уровней.

Алгоритм MD6 предполагает, что перед каждым переходом к более верхнему уровню функций сжатия производится сравнение значения параметра L и номера текущего уровня (до перехода) $level$. Если $level < L$, то алгоритм продолжает параллельные вычисления по древовидной структуре. В противном случае новый уровень $level + 1$ становится последним уровнем структуры алгоритма и вычисления выполняются последовательно.

Таким образом, промежуточные значения L представляют собой комбинацию параллельного и последовательного вариантов алгоритма MD6, что дает промежуточные возможности по распараллеливанию вычислений и промежуточные требования к памяти. Это проиллюстрировано на рис. 1.31, где приведена структура для $L = 1$. Отметим, что и при небольших значениях L алгоритм может оставаться строго параллельным при хешировании сообщений небольших размеров: если 4^L превышает размер сообщения в блоках по 16 слов, то перехода к последовательным вычислениям не происходит.

Рисунок 1.31. Структура алгоритма MD6 при $L = 1$

Можно определить максимальное количество уровней алгоритма, исходя из максимально возможного размера хешируемого сообщения (не более 2^{64} бит) и размера выходного значения функции сжатия (16 слов, т. е. 1024 бита):

$$L_{\max} = \log_4(2^{64} / 2^{10}) = 27.$$

Таким образом, все значения L в диапазоне от 27 до 64 включительно приводят к идентичной структуре алгоритма. При этом результирующие хеш-коды для всех этих значений L оказываются различными, поскольку, как было сказано выше, значение L является параметром каждого вызова функции $f()$.

Нельзя сказать, что наличие альтернативных структур алгоритма и внесение L в параметры функции сжатия выглядит бесспорным решением – представим вполне имеющую право на существование систему, в которой во взаимодействие с использованием алгоритма хеширования MD6 вовлечены как устройства с ограниченными ресурсами (скажем, беспроводные сенсоры – см., например, [246]), так и мультипроцессорный сервер, который занимается сбором и анализом присылаемых сенсорами данных. Выглядит логичным последовательное хеширование на стороне сенсоров с параметром $L = 0$ и параллельное хеширование на стороне сервера с $L = 64$. Однако такой вариант при корректной реализации алгоритма согласно спецификации [214] невозможен, поскольку результаты хеширования одного и того же сообщения с различными значениями L не эквивалентны.

В результате по параметру L можно сделать неоднозначный вывод: полезная гибкость в реализации алгоритма хеширования MD6 может приводить к вынужденной потере совместимости между реализациями алгоритма. Кроме того, в любой системе, в которой выполняется хеширование с различными значениями L , данное значение необходимо каким-либо образом учитывать и передавать для последующего хеширования с тем же значением L на проверяющей стороне.

Использование секретного ключа

Еще один из важных параметров алгоритма – секретный ключ. Его использование позволяет на уровне алгоритма хеширования решить ряд задач, которые обычно решаются различными надстройками над алгоритмом хеширования, в частности:

- ключ может использоваться в качестве «соли» (salt) – случайной или псевдослучайной последовательности данных, которая чаще всего применяется для рандомизации при хешировании паролей (в том числе с целью противодействия словарным атакам [246]); соль не является секретной и обычно хранится вместе с хешированным паролем, поскольку она должна быть доступна при его проверке;
- секретный ключ необходим для различных вариантов ключевого хеширования – в основном для вычисления кодов аутентификации сообщений: MAC, HMAC и т. п.; надстройки аутентификации данных на основе алгоритмов хеширования были подробно рассмотрены ранее.

Секретный ключ K может быть любого размера от нуля (по умолчанию) до 64 байт включительно. Допускается и применение ключей больших размеров; такой ключ необходимо сначала обработать алгоритмом хеширования с нулевым ключом и 512-битовым выходным значением, после чего результат хеширования можно использовать в качестве ключа K . Ключ меньших, чем 64 байта, размеров дополняется до данного размера нулевыми байтами. Как указывают авторы алгоритма MD6 в [214], 64-байтовый ключ может представлять собой и различную комбинацию внутренних полей с различным назначением.

Размер ключа K в байтах обозначается как $keylen$.

Секретный ключ K участвует в вычислениях при каждом вызове функции сжатия алгоритма MD6 следующим образом (см. рис. 1.32):

- значение K является частью входной последовательности функции сжатия;
- значение $keylen$ записывается в дополнительное слово V , которое также является частью входной последовательности функции сжатия.

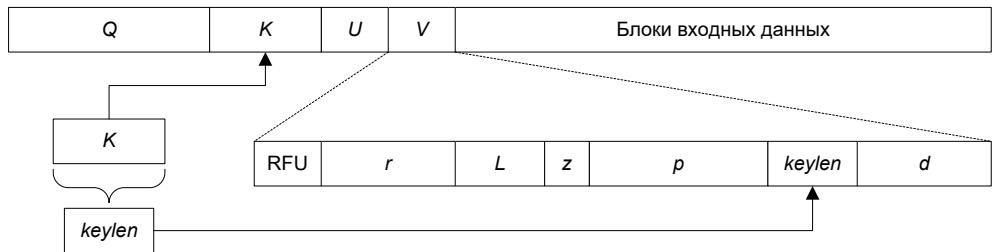


Рисунок 1.32. Ключ как часть входной последовательности функции сжатия алгоритма MD6

Алгоритм MD6 спроектирован таким образом, чтобы никакая информация о значении секретного ключа не была доступна криптоаналитику в результате анализа, например, полученных в результате работы алгоритма хеш-кодов. Для обеспечения отсутствия утечки данных о ключе алгоритм MD6, в числе прочего, устанавливает нижнюю границу количества раундов функции сжатия при использовании ключа, о чем будет сказано далее.

Авторы алгоритма MD6 в [214] утверждают, что криптостойкость некоторых вариантов MAC-надстроек сильно зависит от структуры и криптостойкости нижележащего алгоритма хеширования (что является известным фактом – см., например, [139]). Интегрирование возможности ключевого хеширования в структуру алгоритма решает подобные проблемы с криптостойкостью.

Количество раундов функции сжатия

В базовом варианте алгоритма количество раундов функции сжатия r определено следующим образом:

$$r = 40 + \lfloor d / 4 \rfloor,$$

где d – размер выходного значения алгоритма в битах.

Значением r можно варьировать с целью увеличения криптостойкости алгоритма (путем увеличения r) или с целью увеличения его производительности (путем уменьшения r). При этом если в алгоритме используется ключ K , установлено минимальное количество раундов – 80, что необходимо для обеспечения защиты секретного ключа против возможных атак, направленных на вычисление его значения. Таким образом, при использовании ключа K минимальное количество раундов определяется так:

$$r = \max(80, 40 + \lfloor d / 4 \rfloor).$$

Именование различных вариантов алгоритма

Основные из описанных выше параметров алгоритма (за исключением значения секретного ключа) могут быть явным образом внесены в название конкретного варианта алгоритма. Для единообразия авторы MD6 предлагают пользоваться следующей нотацией (курсивом в названии варианта алгоритма выделены переменные, вместо которых подставляются конкретные значения, а обычным шрифтом – опции названия):

MD6-*d*-*kkeylen*-*LL*-*rr*.

Например, следующий вариант алгоритма:

MD6-512-k64-L2-r192

– представляет собой алгоритм MD6 с 512-битовым выходным значением, 64-байтовым ключом, значением $L = 2$ и 192 раундами.

Любой параметр, кроме d , может быть опущен в названии, если он установлен в значение по умолчанию. Например, такой вариант:

MD6-160-L0

– является 160-битовым алгоритмом MD6 с последовательной структурой и остальными параметрами по умолчанию, т. е. без секретного ключа и с 80 раундами функции сжатия.

Неявные параметры алгоритма

Помимо описанных ранее параметров L , K и r , MD6 использует множество различных величин (константных в базовом варианте алгоритма), которые также могут рассматриваться как параметры алгоритма, т. е. могут быть установлены в различные значения, отличающиеся от значений по умолчанию. К таким неявным параметрам алгоритма относятся следующие величины [214]:

- размер слова – основной единицы измерения размеров данных и констант в алгоритме;
- размер входной последовательности и размер выходного значения функции сжатия алгоритма;

- значение константы Q , являющейся префиксом входной последовательности функции сжатия;
- значения раундовых констант S_j ;
- значения констант $t0...t4$, определяющих позиции обратной связи в итерации функции сжатия;
- значения констант r_j и l_p , определяющих количество битов вращения операнда в итерации функции сжатия.

Неявные параметры алгоритма не вносятся в название его вариантов, поскольку, как отметили авторы алгоритма в [214], название «MD6» подразумевает, что все неявные параметры имеют значения по умолчанию. Тем не менее возможность модификации неявных параметров позволяет создавать на основе MD6 специфические варианты алгоритма хеширования, близкие по структуре к исходному алгоритму.

Результаты криptoанализа

В отличие от широко распространенного алгоритма MD5, криptoанализу которого было посвящено множество работ, алгоритм MD6 распространения практически не получил, в результате чего он привлек несравненно меньшее внимание криptoаналитиков.

Кроме того, через незначительное время после опубликования MD6 – в 2009 г. – авторы алгоритма отзвали его с конкурса по выбору нового стандарта хеширования США SHA-3 по изложенной ими причине, что «в настоящий момент MD6 не соответствует нашим собственным стандартам, соответствие которым мы считаем обязательным для SHA-3» [236], что стало причиной невыхода алгоритма во второй раунд конкурса SHA-3 и способствовало дальнейшему уменьшению внимания к нему.

В результате на текущий момент каких-либо атак на полнораундовый алгоритм MD6 в открытой печати опубликовано не было, но алгоритм не нашел широкого применения.

1.3.2 Алгоритмы семейства RIPEMD

Алгоритмы хеширования, относящиеся к семейству RIPEMD, значительно менее известны и нет так широко распространены по сравнению с рассмотренными ранее алгоритмами MD. Однако они тоже заслуживают подробного рассмотрения, поскольку один из алгоритмов данного семейства – RIPEMD-160 – активно используется в качестве алгоритма хеширования в системе Биткойн и ряде других блокчейн-систем [246] (см. главы 3 и 4).

Данное семейство состоит из пяти алгоритмов хеширования: исходного алгоритма RIPEMD и более поздних алгоритмов RIPEMD-128, RIPEMD-160, RIPEMD-256 и RIPEMD-320, число в названии которых обозначает размерность выходного значения в битах.

Название семейства алгоритмов представляет собой аббревиатуру от RIPE Message Digest (дайджест сообщения в проекте RIPE), где RIPE, в свою очередь, является аббревиатурой от RACE Integrity Primitive Evaluation (оценка примитивов контроля целостности в проекте RACE) – это название подпроекта по формированию портфолио примитивов контроля целостности, включая алгоритмы хеширования, для их последующего использования в рамках проекта RACE [106].

Проект RACE (Research and Development in Advanced Communication Technologies in Europe) был инициирован Европейской Комиссией в середине 1980-х гг. и объединял исследования и разработки с целью внедрения и коммерческого использования общеевропейской системы широкополосной связи IBC (Integrated Broadband Communication – интегрированная широкополосная связь) [246].

Первый алгоритм семейства – RIPEMD – был предложен коллективом авторов из Католического университета г. Львен, Бельгия (Katholieke Universiteit Leuven), в 1991 г. Данный алгоритм был основан на актуальном на тот момент алгоритме MD4, его структура фактически представляла собой параллельно применяющиеся две копии алгоритма MD4 (с различными раундовыми константами и рядом других незначительных модификаций), после применения которых производилось объединение их выходных значений [106].

Алгоритм RIPEMD не нашел широкого применения по причине достаточно быстрого обнаружения в нем ряда потенциальных уязвимостей; впоследствии в 2004 г. была опубликована работа [243], показывающая возможность нахождения коллизий для данного алгоритма.

В результате разработчики алгоритма RIPEMD в 1996 г. предложили новую версию алгоритма RIPEMD – алгоритм RIPEMD-160 [106], который активно применяется до сих пор.

Алгоритм RIPEMD-160

Целью разработки алгоритма RIPEMD-160 было избавление от найденных потенциальных уязвимостей оригинального алгоритма RIPEMD и увеличение размера его выходного значения до 160 бит.

Структура алгоритма

Разбиение входной последовательности на блоки и дополнение последнего блока выполняются полностью аналогично алгоритму MD4, который был подробно описан ранее.

Вместо четырех 32-битовых регистров $A \dots D$ алгоритма MD4 (содержащих текущее состояние алгоритма) в RIPEMD-160 используется 5 регистров $A \dots E$ такого же размера. Данные регистры инициализируются начальными значениями согласно табл. 1.12 (приведены шестнадцатеричные значения) [106].

Таблица 1.12. Начальное заполнение регистров алгоритма RIPEMD-160

Регистр	Исходное значение
<i>A</i>	67452301
<i>B</i>	EFCDAB89
<i>C</i>	98BADCFE
<i>D</i>	10325476
<i>E</i>	C3D2E1F0

Над каждым из 512-битовых блоков входных данных после их дополнения выполняются следующие действия:

1. Блок входных данных представляется в виде 32-битовых слов $M[0] \dots M[15]$.
2. Текущее содержимое регистров $A \dots E$ копируется в два набора временных переменных: $a \dots e$ и $a' \dots e'$. Поскольку алгоритм RIPEMD-160 фактически выполняет два потока преобразований, он использует два набора временных переменных, каждый из которых применяется в одном из потоков.
3. Выполняются 80 итераций (см. рис. 1.33), в каждой из которых происходит обновление переменных $a \dots e$ с участием одного из слов блока входных данных:

$$t = (a + f_i(b, c, d) + M[N_i] + K_i \bmod 2^{32}) \ll s_i + e \bmod 2^{32};$$

$$a = e;$$

$$e = d;$$

$$d = c \ll 10;$$

$$c = b;$$

$$b = t,$$

где:

- i – номер итерации, $i = 0 \dots 79$ (каждые 16 итераций формируют один из пяти раундов алгоритма);
- t – временная 32-битовая переменная;
- $f_i()$ – одна из следующих раундовых функций:

Таблица 1.13. Раундовые функции первого потока преобразований алгоритма RIPEMD-160

Итерации	$f_i()$
0...15	$f(x, y, z) = x \oplus y \oplus z$
16...31	$f(x, y, z) = (x \& y) (\sim x \& z)$
32...47	$f(x, y, z) = (x \sim y) \oplus z$
48...63	$f(x, y, z) = (x \& z) (y \& \sim z)$
64...79	$f(x, y, z) = x \oplus (y \sim z)$

- N_i – константа, определяющая 32-битовое слово блока входных данных, используемое в i -й итерации в соответствии со следующей таблицей, в ячейках которой последовательно перечислены константы N_i для $i = 0 \dots 79$:

Таблица 1.14. Константы выбора слов блока входных данных первого потока преобразования алгоритма RIPEMD-160

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2
4	0	5	9	7	12	2	10	14	1	3	8	11	6	15	13

- K_i – одна из следующих раундовых констант (приведены шестнадцатеричные значения):

Таблица 1.15. Раундовые константы первого потока преобразований алгоритма RIPEMD-160

Итерации	K_i
0...15	0
16...31	5A827999
32...47	6ED9EBA1
48...63	8F1BBCDC
64...79	A953FD4E

- s_i – константа, определяющая количество битов в операции вращения (циклического сдвига) в соответствии со следующей таблицей (в ячейках таблицы последовательно перечислены константы s_i для $i = 0 \dots 79$):

Таблица 1.16. Константы, определяющие количество битов вращения первого потока преобразований алгоритма RIPEMD-160

11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
11	13	6	7	14	9	13	15	14	8	13	6	5	12	7	5
11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12
9	15	5	11	6	8	13	12	5	12	13	14	11	8	5	6

4. Выполняются аналогичные 80 итераций, в каждой из которых происходит обновление переменных $a' \dots e'$ со следующими отличиями от итераций первого потока преобразований:

- в другом порядке используются раундовые функции (обозначим их $f_i'()$):

Таблица 1.17. Раундовые функции второго потока преобразований алгоритма RIPEMD-160

Итерации	$f'_i()$
0...15	$f'(x, y, z) = x \oplus (y \mid \sim z)$
16...31	$f'(x, y, z) = (x \& z) \mid (y \& \sim z)$
32...47	$f'(x, y, z) = (x \mid \sim y) \oplus z$
48...63	$f'(x, y, z) = (x \& y) \mid (\sim x \& z)$
64...79	$f'(x, y, z) = x \oplus y \oplus z$

- в другом порядке производится выбор слов входного блока данных – в соответствии со следующей таблицей:

Таблица 1.18. Константы выбора слов блока входных данных второго потока преобразований алгоритма RIPEMD-160

5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14
12	15	10	4	1	5	8	7	6	2	13	14	0	3	9	11

- используется другой набор раундовых констант K'_i (приведены шестнадцатеричные значения):

Таблица 1.19. Раундовые константы второго потока преобразований алгоритма RIPEMD-160

Итерации	K'_i
0...15	50A28BE6
16...31	5C4DD124
32...47	6D703EF3
48...63	7A6D76E9
64...79	0

- используются другие константы, определяющие количество битов в операции вращения (s'_i):

Таблица 1.20. Константы, определяющие количество битов вращения второго потока преобразований алгоритма RIPEMD-160

5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14
12	15	10	4	1	5	8	7	6	2	13	14	0	3	9	11

5. Текущие значения переменных $A \dots E$ обновляются с участием значений переменных $a \dots e$ и $a' \dots e'$, полученных в результате выполнения описанных выше итераций:

$$t = B + c + d' \bmod 2^{32};$$

$$B = C + d + e' \bmod 2^{32};$$

$$C = D + e + a' \bmod 2^{32};$$

$$D = E + a + b' \bmod 2^{32};$$

$$E = A + b + c' \bmod 2^{32};$$

$$A = t.$$

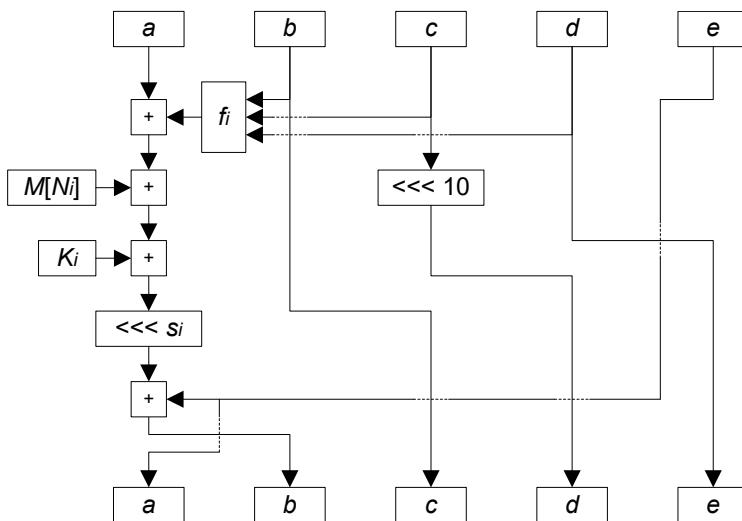


Рисунок 1.33. Итерация алгоритма RIPEMD-160

На этом обработка блока входных данных завершается. Описанная выше процедура обработки схематично изображена на рис. 1.34.

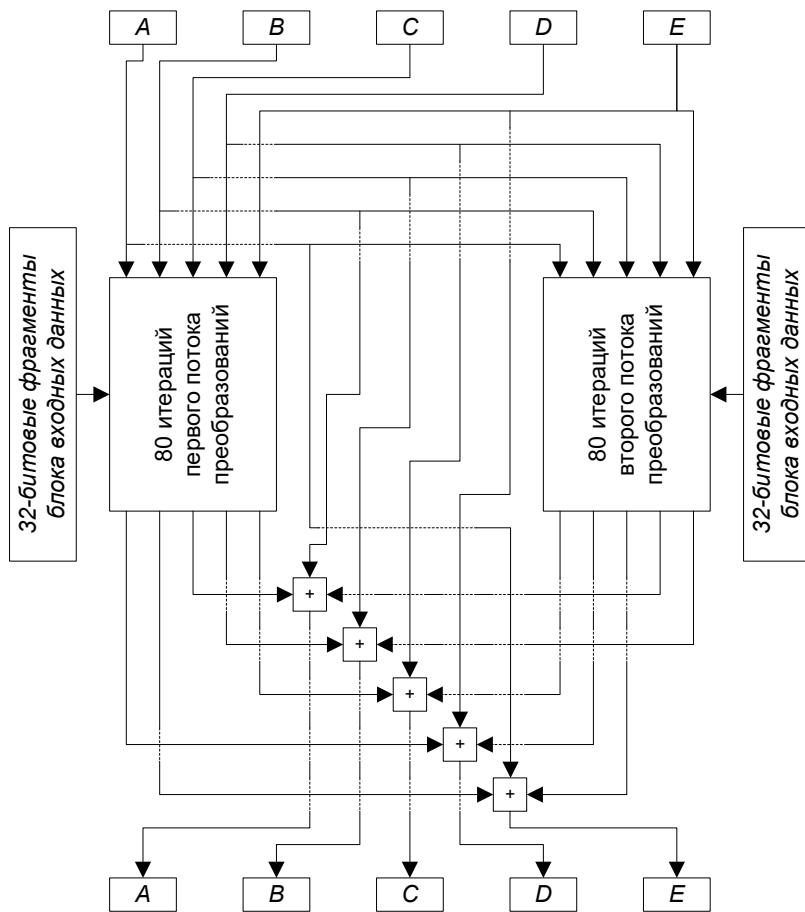


Рисунок 1.34. Схема обработки блока входных данных в алгоритме RIPEMD-160

Выходное значение алгоритма хеширования RIPEMD-160 формируется в результате конкатенации значений регистров $A \dots E$, полученных после обработки последнего блока дополненного сообщения.

Альтернативное представление ряда элементов структуры

В реализациях алгоритма, предназначенных для использования в устройствах с крайне ограниченными ресурсами, может показаться затратным хранение в какой-либо энергонезависимой памяти всех приведенных выше таблиц констант.

Поскольку многие из приведенных выше констант являются результатом определенных преобразований, в подобных реализациях вместо их хранения может быть более оптимальным вычисление констант по мере необходимости – непосредственно перед их использованием.

В частности, приведенные выше раундовые константы K_i и K'_i могут быть вычислены следующим образом (дробная часть после вычисления отбрасывается):

Таблица 1.21. Вычисление раундовых констант алгоритма RIPEMD-160

Итерации	K_i	K'_i
0...15	0	$2^{30} * \sqrt[3]{2}$
16...31	$2^{30} * \sqrt{2}$	$2^{30} * \sqrt[3]{3}$
32...47	$2^{30} * \sqrt{3}$	$2^{30} * \sqrt[3]{5}$
48...63	$2^{30} * \sqrt{5}$	$2^{30} * \sqrt[3]{7}$
64...79	$2^{30} * \sqrt{7}$	0

Помимо раундовых констант, константы N_i , определяющие выбираемые в итерациях слова блока входных данных, также являются вычисляемыми.

Для их вычисления используется перестановка ρ_j , определенная следующим образом:

Таблица 1.22. Перестановка слов блока входных данных в алгоритме RIPEMD-160

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ρ_j	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

В данной таблице j определяет номер слова блока входных данных при текущем расположении слов, а ρ_j – новое положение слова j после применения перестановки.

Помимо перестановки ρ_j , определена также перестановка π_j :

$$\pi_j = 9j + 5 \bmod 16.$$

Перестановки ρ_j и π_j определяют порядок использования слов блока входных данных в соответствии со следующей таблицей:

Таблица 1.23. Порядок применения перестановок слов блока входных данных в алгоритме RIPEMD-160

Итерации	Первый поток преобразований	Второй поток преобразований
0...15	i	π
16...31	ρ	$\rho\pi$
32...47	ρ^2	$\rho^2\pi$
48...63	ρ^3	$\rho^3\pi$
64...79	ρ^4	$\rho^4\pi$

Быстродействие и результаты криптоанализа

За счет фактически дублирования преобразований по сравнению с близкими аналогами алгоритм RIPEMD-160 является значительно более медленным. Замеры быстродействия, проведенные авторами данного алгоритма

и опубликованные в [106], показали, что RIPEMD-160 является в 4 раза более медленным по сравнению с MD4, почти в 3 раза медленнее MD5 и на 15 % медленнее SHA-1 (алгоритмы семейства SHA подробно описаны далее). При этом отметим, что все перечисленные алгоритмы, с которыми RIPEMD-160 сравнивался по быстродействию, успешно атакованы много лет назад.

Что касается криптоанализа, то алгоритм RIPEMD-160 как основной алгоритм хеширования, используемый в платформе Биткойн, привлекает пристальное внимание криптоаналитиков. Однако на текущий момент какие-либо атаки на полноразрядовую версию данного алгоритма не получили широкой известности.

Следовательно, уменьшение быстродействия можно считать оправданным.

Алгоритм RIPEMD-128

Одновременно с алгоритмом RIPEMD-160 в работе [106] был предложен алгоритм RIPEMD-128 со 128-битовым выходным значением для возможной замены алгоритма MD4 на более криптостойкий алгоритм с тем же размером хеш-значения.

По своей общей структуре RIPEMD-128 аналогичен алгоритму RIPEMD-160 – он также выполняет два потока преобразований, но, как и алгоритм MD4, использует 4 регистра состояния по 32 бита: $A \dots D$.

Основные отличия между данными алгоритмами сводятся к следующим:

1. Выполняются только 4 раунда преобразований (64 итерации).
2. В каждой итерации выполняются следующие действия:

$$\begin{aligned} t = (a + f_i(b, c, d) + M[N_i] + K_i \bmod 2^{32}) \lll s_i; \\ a = d; \\ d = c; \\ c = b; \\ b = t. \end{aligned}$$

Итерация алгоритма RIPEMD-128 схематично изображена на рис. 1.35.

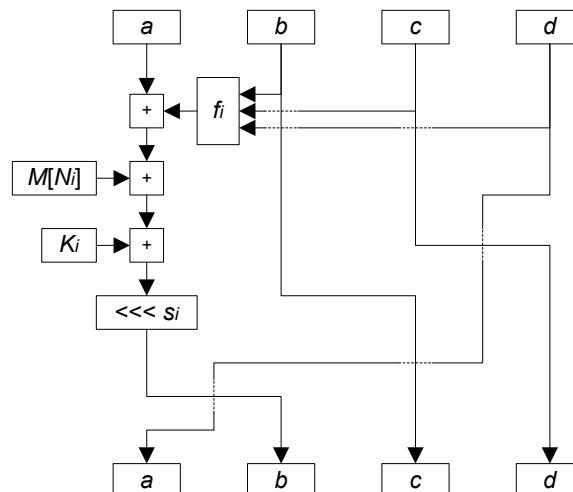


Рисунок 1.35. Итерация алгоритма RIPEMD-128

3. Используется подмножество описанных ранее для алгоритма RIPEMD-160 раундовых функций и констант в соответствии с таблицами:

Таблица 1.24. Раундовые функции алгоритма RIPEMD-128

Итерации	$f_i()$	$f'_i()$
0...15	$f(x, y, z) = x \oplus y \oplus z$	$f(x, y, z) = (x \& z) \mid (y \& \sim z)$
16...31	$f(x, y, z) = (x \& y) \mid (\sim x \& z)$	$f(x, y, z) = (x \mid \sim y) \oplus z$
32...47	$f(x, y, z) = (x \mid \sim y) \oplus z$	$f(x, y, z) = (x \& y) \mid (\sim x \& z)$
48...63	$f(x, y, z) = (x \& z) \mid (y \& \sim z)$	$f(x, y, z) = x \oplus y \oplus z$

Таблица 1.25. Вычисление раундовых констант алгоритма RIPEMD-128

Итерации	K_i	K'_i
0...15	0	$2^{30} * \sqrt[3]{2}$
16...31	$2^{30} * \sqrt{2}$	$2^{30} * \sqrt[3]{3}$
32...47	$2^{30} * \sqrt{3}$	$2^{30} * \sqrt[3]{5}$
48...63	$2^{30} * \sqrt{5}$	0

Алгоритмы RIPEMD-256 и RIPEMD-320

Данные алгоритмы семейства RIPEMD были предложены для использования в случаях, когда требуется выходное значение удвоенного размера.

Алгоритмы RIPEMD-256 и RIPEMD-320 построены на незначительной модификации лежащих в их основе алгоритмов RIPEMD-128 и RIPEMD-160 соответственно. Данная модификация состоит в следующем:

- используется удвоенное количество регистров для хранения состояния алгоритмов: 8 для RIPEMD-256 ($A \dots D$ и $A' \dots D'$) и 10 для RIPEMD-320 ($A \dots E$ и $A' \dots E'$);
- исключаются операции по объединению результатов применения двух потоков преобразований – значения переменных после выполнения преобразований (например, $a \dots d$ и $a' \dots d'$ для RIPEMD-256) не объединяются, а только накладываются на регистры ($A \dots D$ и $A' \dots D'$ для RIPEMD-256) операцией сложения по модулю 2^{32} ;
- после выполнения каждого раунда преобразований (т. е. каждых 16 итераций) пары регистров из первого и второго потоков преобразований меняются между собой (a и a' после первого раунда, b и b' – после второго и т. д.).

Процедура обработки одного блока входных данных на примере алгоритма RIPEMD-256 схематично изображена на рис. 1.36.

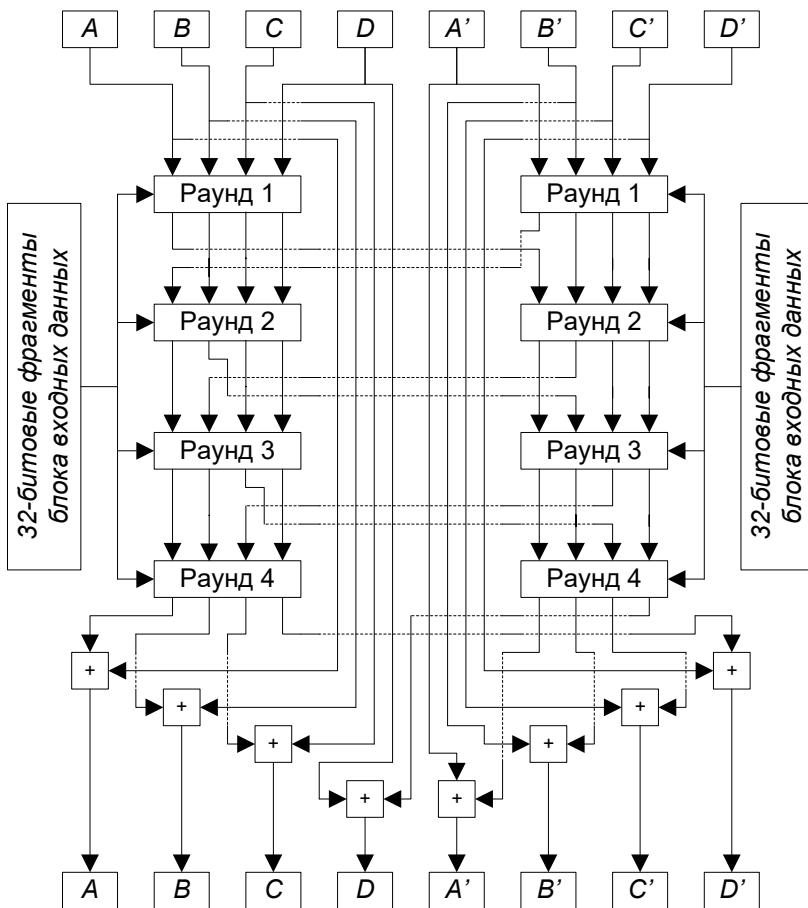


Рисунок 1.36. Схема обработки блока входных данных в алгоритме RIPEMD-256

Следует отметить, что увеличения уровня криптостойкости таких алгоритмов с удвоенным по размеру выходным значением по сравнению с исходными алгоритмами не происходит [106].

Алгоритмы RIPEMD-256 и RIPEMD-320 являются шаблонными; в их описании [106] не приведены конкретные значения ряда параметров алгоритма, включая начальное заполнение регистров.

1.3.3 Алгоритмы семейства SHA

Алгоритмы семейства SHA (Secure Hash Algorithm – алгоритм безопасного хеширования) в разные годы были (и некоторые из них остаются в настоящее время) стандартами хеширования США, что предопределило их широчайшее использование в мире. Алгоритмы данного семейства подробно описаны далее.

Алгоритм SHA-0

Алгоритм SHA был предложен Национальным институтом стандартов и технологий США (NIST – National Institute of Standards and Technology) в январе 1992 г.

в качестве алгоритма хеширования, лежащего в основе стандарта хеширования США SHS (Secure Hash Standard – стандарт безопасного хеширования) [89]. Предполагалось, что стандарт SHS должен использоваться в паре с разработанным в 1991 г. стандартом США на электронную подпись DSA (Digital Signature Algorithm – алгоритм цифровой подписи) – DSS (Digital Signature Standard – стандарт цифровой подписи) [130].

Стандартом SHS предписывалось, что все министерства и Федеральные агентства США обязаны применять SHA при необходимости обеспечения целостности несекретной информации. Коммерческим организациям NIST рекомендовал использовать SHA в тех же целях [89, 125].

Первый вариант стандарта SHS с описанием алгоритма SHA вышел в мае 1993 г. [125]. Как было сказано в стандарте SHS, при разработке SHA за основу был взят рассмотренный ранее алгоритм MD4.

Впоследствии, когда появился следующий алгоритм семейства SHA (т. е. алгоритм SHA-1), за первоначальным алгоритмом SHA закрепилось название SHA-0.

Структура алгоритма

Алгоритм SHA вычисляет 160-битовый хеш-код от входного блока данных переменного размера – от 0 до ($2^{64} - 1$) бит. Последовательность работы алгоритма приведена далее.

Прежде всего входные данные разбиваются на блоки размером по 512 бит каждый. Последний блок всегда дополняется до 512-битового размера следующим образом:

- сначала добавляется единичный бит;
- затем – нулевые биты до размера 448 бит (т. е. 512 – 64);
- наконец, к последовательности добавляется 64-битовое значение, равное размеру в битах исходной последовательности входных данных до дополнения (сначала добавляется старшее 32-битовое слово данного значения, затем – младшее).

Если изначально размер последнего блока равен 448 бит (или больше), дополнение выполняется все равно, при этом количество блоков увеличивается на один.

Затем выполняется инициализация пяти 32-битовых регистров A, B, C, D и E согласно следующей таблице (указаны шестнадцатеричные значения):

Таблица 1.26. Начальное заполнение регистров алгоритма SHA

Регистр	Исходное значение
A	67452301
B	EFCDA89
C	98BADCFE
D	10325476
E	C3D2E1F0

Над каждым из 512-битовых блоков дополненных входных данных выполняются следующие преобразования:

1. 512-битовый блок (представляемый в виде 32-битовых слов $M_0 \dots M_{15}$) расширяется до размера в 2560 бит (т. е. 80 слов по 32 бита $W_0 \dots W_{79}$) следующим образом (см. рис. 1.37):

$$W_n = M_n \text{ для } n = 0 \dots 15;$$

$$W_n = W_{n-3} \oplus W_{n-8} \oplus W_{n-14} \oplus W_{n-16} \text{ для } n > 15.$$

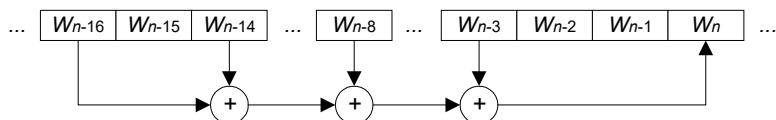


Рисунок 1.37. Расширение обрабатываемого блока в алгоритме SHA

2. Содержимое регистров $A \dots E$ копируется во временные переменные $a \dots e$.

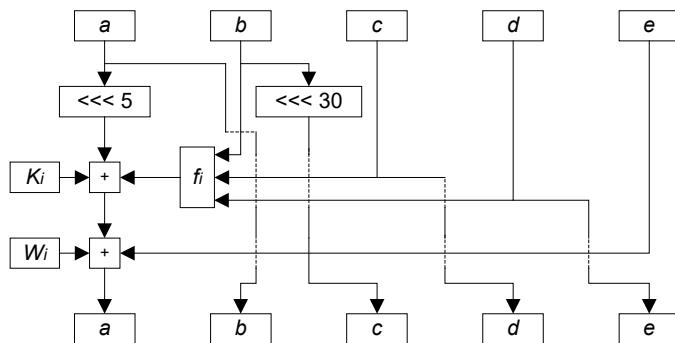


Рисунок 1.38. Итерация алгоритма SHA

3. Выполняется 80 итераций преобразований (см. рис. 1.38), в каждой из которых модифицируются переменные $a \dots e$ с участием одного из фрагментов расширенного блока W_i (где i – номер текущей итерации от 0 до 79):

$$t = (a \lll 5) + f_i(b, c, d) + e + W_i + K_i \bmod 2^{32};$$

$$e = d;$$

$$d = c;$$

$$c = b \lll 30;$$

$$b = a;$$

$$a = t.$$

Значения модифицирующих констант K_i приведены в таблице (в шестнадцатеричном виде):

Таблица 1.27. Модифицирующие константы алгоритма SHA

Итерации	K_i
0...19	5A827999
20...39	6ED9EBA1
40...59	8F1BBCDC
60...79	CA62C1D6

Используемые в итерациях алгоритма SHA функции $f_i()$ определены следующим образом:

Таблица 1.28. Функции итераций алгоритма SHA

Итерации	$f_i()$
0...19	$f(x, y, z) = (x \& y) (\sim x \& z)$
20...39, 60...79	$f(x, y, z) = x \oplus y \oplus z$
40...59	$f(x, y, z) = (x \& y) (x \& z) (y \& z)$

4. Значения регистров $A...E$ складываются по модулю 2^{32} с полученными значениями переменных $a...e$ соответственно.

Итоговый 160-битовый хеш-код – результат конкатенации регистров $A...E$.

Поскольку при реализации SHA в устройствах, обладающих недостаточными ресурсами, требования SHA к оперативной памяти могли показаться черезчур большими, стандарт SHS также описывал альтернативный вариант алгоритма, приводящий к тому же результату, но требующий значительно меньше оперативной памяти (в ущерб скорости вычислений). Суть данного варианта состоит в динамическом (по мере необходимости) вычислении значений $W_{16}...W_{79}$ [89].

Результаты криптоанализа

В то время, когда SHA-0 был стандартом хеширования, не было опубликовано каких-либо криптоаналитических атак на данный алгоритм. В частности, в [53] было сказано, что сведения об успешных криптографических атаках на алгоритм SHA отсутствуют.

Впоследствии было опубликовано достаточно большое количество атак на алгоритм SHA-0, в результате которых данный алгоритм считается полностью взломанным. В частности, трудоемкость успешного поиска коллизий для SHA-0 составляет всего 2^{33} операций [169, 196].

Алгоритм SHA-1

Меньше, чем через два года после появления первого варианта стандарта SHS [125], в 1995 г. стандарт SHS был заменен более новым вариантом [126]. Данный вариант стандарта ввел алгоритм хеширования SHA-1 вместо SHA (который с этого момента стал называться SHA-0).

Алгоритмы SHA-0 и SHA-1 почти идентичны и различаются только способом формирования на основе обрабатываемого блока 80 слов $W_0 \dots W_{79}$: приведенная выше формула вычисления W_n (для алгоритма SHA) дополняется циклическим сдвигом значения полученного слова на 1 бит влево (см. рис. 1.39):

$$W_n = (W_{n-5} \oplus W_{n-8} \oplus W_{n-14} \oplus W_{n-16}) \lll 1.$$

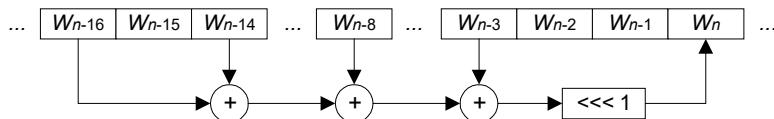


Рисунок 1.39. Расширение обрабатываемого блока в алгоритме SHA-1

Стандарт [126] никак не поясняет причины данной модификации алгоритма, ограничиваясь замечанием, что данная ревизия увеличивает безопасность, обеспечиваемую стандартом. Впоследствии была опубликована работа [73], в которой показано, что вращение на 1 бит в процедуре расширения хешируемого блока существенно сильнее перемешивает биты блока при его расширении, чем в SHA-0. Результатом этого является тот факт, что атаки методом дифференциального криптоанализа против SHA-1 существенно менее эффективны, чем против SHA-0.

Результаты криптоанализа

Криптоанализ алгоритмов SHA-0 и SHA-1 во многом шел параллельно – новые результаты анализа SHA-0 криптологи пытались применить к SHA-1, и наоборот. Однако, поскольку SHA-1 существенно больше, чем SHA-0, пробыл в качестве стандарта хеширования США, его криптоанализу было посвящено заметно больше работ.

В числе прочего было опубликовано несколько атак на полнораундовые версии алгоритма, в результате которых алгоритм SHA-1 можно считать взломанным. В частности, трудоемкость поиска коллизий для алгоритма SHA-1 составляет 2^{52} операций [171].

Семейство алгоритмов SHA-2

Семейство алгоритмов SHA-2 было определено в новой версии стандарта SHS [127], который вышел в 2002 г. (введен в действие с 1 февраля 2003 г.) на смену предыдущей редакции [126].

Данный стандарт не запрещает использование алгоритма SHA-1, а только добавляет алгоритмы SHA-2, способные вычислять хеш-коды большего размера, чем SHA-1.

Стоит отметить, что хотя «SHA-2» является общеупотребительным названием данного семейства алгоритмов, само это название нигде официально не стандартизовано [223]; в [127] используется наименование алгоритмов SHA- n , где n – размер выходного значения алгоритма в битах.

Изначально данная версия стандарта SHS описывала три алгоритма семейства SHA-2: SHA-256, SHA-384 и SHA-512. Подробное описание данных алгоритмов приведено далее.

SHA-256

Так же, как и алгоритмы SHA-0 и SHA-1, алгоритм SHA-256 вычисляет хеш-код входного блока данных переменного размера от 0 до $(2^{64} - 1)$ бит включительно. Полностью аналогично алгоритмам SHA-0 и SHA-1 выполняются разбиение входных данных на 512-битовые блоки и дополнение последнего блока.

Однако вместо пяти 32-битовых регистров в SHA-256 используются 8 регистров $A \dots H$, которые инициализируются следующими константами (указаны шестнадцатеричные значения):

Таблица 1.29. Начальное заполнение регистров алгоритма SHA-256

Регистр	Исходное значение
A	6A09E667
B	BB67AE85
C	3C6EF372
D	A54FF53A
E	510E527F
F	9B05688C
G	1F83D9AB
H	5BE0CD19

Каждый из 512-битовых блоков дополненных входных данных обрабатывается следующим образом:

1. 512-битовый блок представляется в виде 16 слов $M_0 \dots M_{15}$ по 32 бита и расширяется до 64 32-битовых слов $W_0 \dots W_{63}$ следующим образом (см. рис. 1.40):

$$W_n = M_n \text{ для } n = 0 \dots 15;$$

$$W_n = \text{Sig}_{1,256}(W_{n-2}) + W_{n-7} + \text{Sig}_{0,256}(W_{n-15}) + W_{n-16} \bmod 2^{32} \text{ для } n > 15.$$

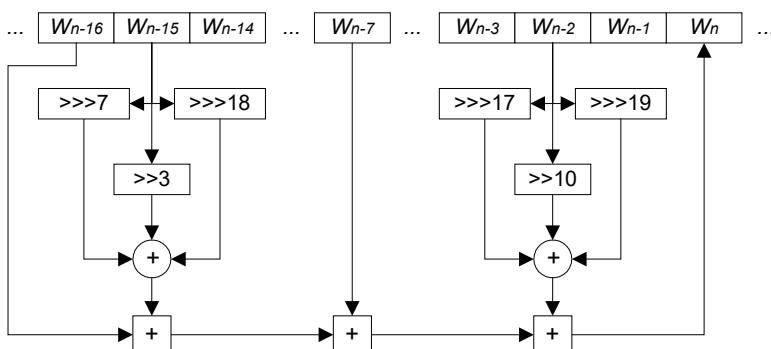


Рисунок 1.40. Расширение обрабатываемого блока в алгоритме SHA-256

Используемые здесь функции $\text{Sig}_{0,256}()$ и $\text{Sig}_{1,256}()$ определены так:

$$\text{Sig}_{0,256}(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3);$$

$$\text{Sig}_{1,256}(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10).$$

Здесь и далее сдвиг значения аргумента вправо (не циклический) обозначен как \gg , а \ggg – циклический сдвиг (вращение) вправо.

2. Содержимое регистров $A \dots H$ копируется во временные переменные $a \dots h$.
3. Выполняется 64 итерации преобразований (см. рис. 1.41), в каждой из которых с участием значения одного из фрагментов расширенного блока W_i ($i = 0 \dots 63$) модифицируются переменные $a \dots h$:

$$T_1 = h + \text{Sum}_{1,256}(e) + \text{Ch}(e, f, g) + W_i + K_{i,256} \bmod 2^{32};$$

$$T_2 = \text{Sum}_{0,256}(a) + \text{Maj}(a, b, c) \bmod 2^{32};$$

$$h = g;$$

$$g = f;$$

$$f = e;$$

$$e = d + T_1 \bmod 2^{32};$$

$$d = c;$$

$$c = b;$$

$$b = a;$$

$$a = T_1 + T_2 \bmod 2^{32},$$

где T_1 и T_2 – временные переменные, а участвующие в вычислениях функции определены следующим образом:

$$\text{Sum}_{0,256}(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22);$$

$$\text{Sum}_{1,256}(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25);$$

$$\text{Ch}(x, y, z) = (x \& y) \oplus (x \& z);$$

$$\text{Maj}(x, y, z) = (x \& y) \oplus (x \& z) \oplus (y \& z).$$

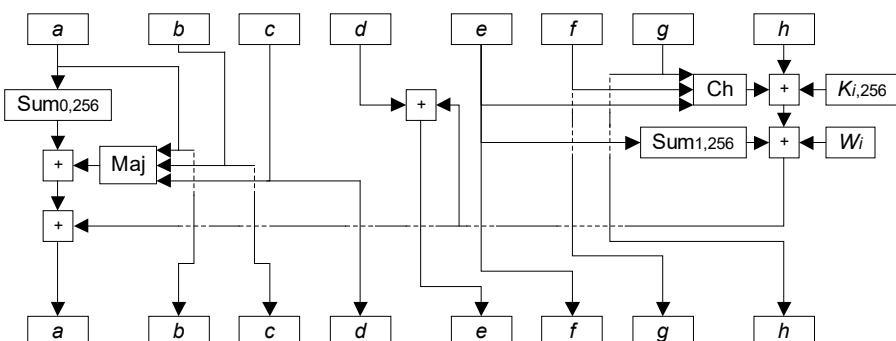


Рисунок 1.41. Итерация алгоритма SHA-256

В отличие от алгоритмов SHA-0 и SHA-1, в алгоритме SHA-256 определены различные константы $K_{i,256}$ для каждого раунда; раундовые константы приведены в приложении 1.

4. Значения регистров $A \dots H$ складываются по модулю 2^{32} с полученными значениями переменных $a \dots h$ соответственно.

Итоговый 256-битовый хеш-код – результат конкатенации регистров $A \dots H$. **SHA-512**

Алгоритм SHA-512 вычисляет хеш-код входного блока данных переменного размера от 0 до $(2^{128} - 1)$ бит включительно. В отличие от описанных ранее алгоритмов семейства SHA, алгоритм SHA-512 разбивает входные данные на 1024-битовые (а не на 512-битовые) блоки.

Дополнение последнего блока данных выполняется следующим образом:

- добавляется единичный бит;
- затем – нулевые биты до размера 896 бит (т. е. $1024 - 128$);
- к последовательности добавляется 128-битовое значение, равное разме- ру в битах исходной последовательности входных данных до дополне-ния (сначала добавляется старшее 64-битовое слово данного значения, затем – младшее).

Дополнение выполняется всегда; при необходимости количество блоков увеличивается на один.

Как и SHA-256, алгоритм SHA-512 использует 8 регистров $A \dots H$, однако данные регистры являются 64-битовыми. Перед началом вычисления хеш-кода они инициализируются следующими константами (указаны шестнадцатеричные значения):

Таблица 1.30. Начальное заполнение регистров алгоритма SHA-512

Регистр	Исходное значение
<i>A</i>	6A09E667F3BCC908
<i>B</i>	BB67AE8584CAA73B
<i>C</i>	3C6EF372FE94F82B
<i>D</i>	A54FF53A5F1D36F1
<i>E</i>	510E527FADE682D1
<i>F</i>	9B05688C2B3E6C1F
<i>G</i>	1F83D9ABFB41BD6B
<i>H</i>	5BE0CD19137E2179

Каждый из блоков дополненных входных данных обрабатывается следую- щим образом:

1. 1024-битовый блок представляется в виде 16 слов $M_0 \dots M_{15}$ по 64 бита и расширяется до 80 64-битовых слов $W_0 \dots W_{79}$ следующим образом (см. рис. 1.42):

$$W_n = M_n \text{ для } n = 0 \dots 15; \\ W_n = \text{Sig}_{1,512}(W_{n-2}) + W_{n-7} + \text{Sig}_{0,512}(W_{n-15}) + W_{n-16} \bmod 2^{64} \text{ для } n > 15.$$

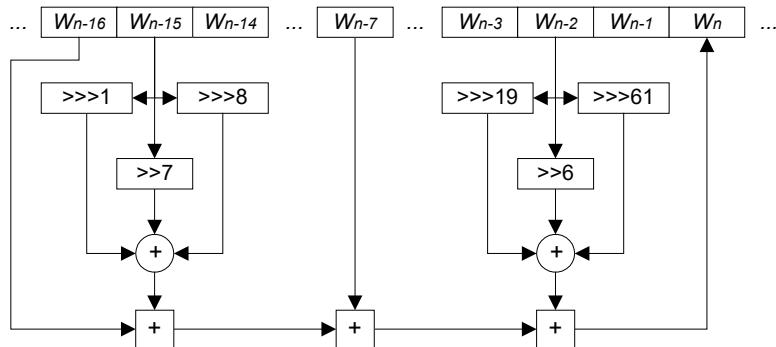


Рисунок 1.42. Расширение обрабатываемого блока в алгоритме SHA-512

Используемые здесь функции $\text{Sig}_{0,512}()$ и $\text{Sig}_{1,512}()$ определены так:

$$\text{Sig}_{0,512}(x) = (x \ggg 1) \oplus (x \ggg 8) \oplus (x \gg 7);$$

$$\text{Sig}_{1,512}(x) = (x \ggg 19) \oplus (x \gg 61) \oplus (x \gg 6).$$

2. Содержимое регистров $A \dots H$ копируется во временные переменные $a \dots h$.
3. Выполняется 80 итераций (см. рис. 1.43), в каждой из которых следующим образом модифицируются переменные $a \dots h$:

$$T_1 = h + \text{Sum}_{1,512}(e) + \text{Ch}(e, f, g) + W_i + K_{i,512} \bmod 2^{64};$$

$$T_2 = \text{Sum}_{0,512}(a) + \text{Maj}(a, b, c) \bmod 2^{64};$$

$$h = g;$$

$$g = f;$$

$$f = e;$$

$$e = d + T_1 \bmod 2^{64};$$

$$d = c;$$

$$c = b;$$

$$b = a;$$

$$a = T_1 + T_2 \bmod 2^{64}.$$

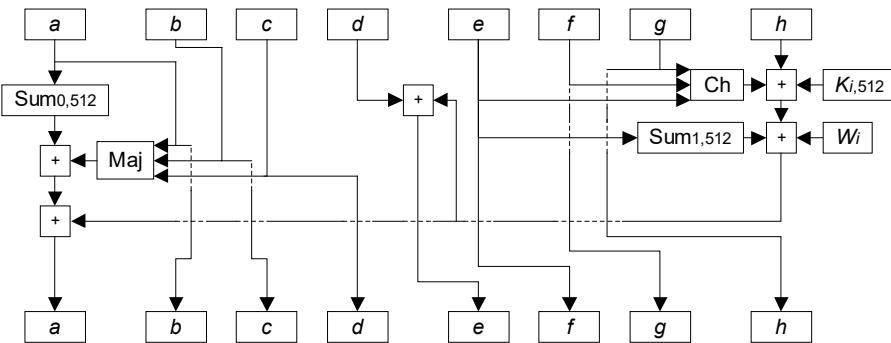


Рисунок 1.43. Итерация алгоритма SHA-512

Итерация SHA-512 весьма похожа на таковую в алгоритме SHA-256 с учетом вычислений по модулю 2^{64} , а также изменений в используемых функциях $\text{Sum}_{x,y}$:

$$\text{Sum}_{0,512}(x) = (x \ggg 28) \oplus (x \ggg 34) \oplus (x \ggg 39);$$

$$\text{Sum}_{1,512}(x) = (x \ggg 14) \oplus (x \ggg 18) \oplus (x \ggg 41).$$

Функции $\text{Ch}()$ и $\text{Maj}()$ аналогичны описанным ранее.

Константы $K_{i,512}$, различные для каждого раунда, приведены в приложении 1.

4. Значения регистров $A \dots H$ складываются по модулю 2^{64} с полученными значениями переменных $a \dots h$ соответственно.

Итоговый 512-битовый хеш-код – результат конкатенации 64-битовых регистров $A \dots H$.

SHA-384

Алгоритм SHA-384 выполняет практически те же преобразования, что и описанный выше алгоритм SHA-512. Отличия данных алгоритмов состоят в следующем:

1. Регистры $A \dots H$ инициализируются другим набором констант (указаны шестнадцатеричные значения):

Таблица 1.31. Начальное заполнение регистров алгоритма SHA-384

Регистр	Исходное значение
A	CBBB9D5DC1059ED8
B	629A292A367CD507
C	9159015A3070DD17
D	152FECD8F70E5939
E	67332667FFC00B31
F	8EB44A8768581511
G	DB0C2E0D64F98FA7
H	47B5481DBEFA4FA4

2. В качестве итогового 384-битового хеш-кода используется результат конкатенации только первых шести 64-битовых регистров, а именно регистров $A \dots F$.

SHA-224

25 февраля 2004 г. вышло дополнение к стандарту FIPS 180-2 [128], включающее еще один алгоритм хеширования семейства SHA-2 – SHA-224.

Кроме того, дополнение к стандарту устанавливает, что в случае необходимости использования хеш-кодов некоторого размера, не предусмотренного алгоритмом SHA-1 и алгоритмами семейства SHA-2 (но меньшего 512 бит), следует использовать алгоритм семейства SHA-2 с ближайшим выходным значением (больше требуемого), после чего брать требуемое количество левых битов результата. Однако такое применение стандарта разрешено только для совместимости со старыми реализациями, в которых предусмотрен фиксированный размер хеш-кода. Данное «усечение» выходного значения может отрицательно повлиять на безопасность такой реализации и не разрешено для ряда применений.

SHA-224 соотносится с алгоритмом SHA-256 так же, как соотносятся описанные ранее алгоритмы SHA-384 и SHA-512. Отличия SHA-224 от SHA-256 состоят в следующем:

1. Регистры $A \dots H$ инициализируются другими константами (указаны шестнадцатеричные значения):

Таблица 1.32. Начальное заполнение регистров алгоритма SHA-224

Регистр	Исходное значение
<i>A</i>	C1059ED8
<i>B</i>	367CD507
<i>C</i>	3070DD17
<i>D</i>	F70E5939
<i>E</i>	FFC00B31
<i>F</i>	68581511
<i>G</i>	64F98FA7
<i>H</i>	BEFA4FA4

2. В качестве итогового 224-битового хеш-кода используется результат конкатенации только первых семи 32-битовых регистров, а именно регистров $A \dots G$.

SHA-512/224 и SHA-512/256

В более новой версии стандарта SHS [129] были введены два новых алгоритма семейства SHA-2: SHA-512/224 и SHA-512/256.

Оба этих алгоритма образованы фактически усечением выходного значения алгоритма хеширования SHA-512 (аналогично SHA-384), но с другими (различ-

ными между собой и отличающимися от SHA-384 и SHA-512) начальными значениями регистров.

Результаты криптоанализа

На текущий момент не опубликовано атак в части поиска коллизий или прообразов на какой-либо из вариантов полнораундовых алгоритмов SHA-2. Однако данное семейство алгоритмов подвержено некоторым менее значительным проблемам, в частности возможности нахождения коллизий путем расширения хешируемых данных [246].

Кроме того, в экспертном сообществе достаточно давно существуют сомнения относительно криптостойкости алгоритмов семейства SHA-2 по причине их значительного сходства со взломанными алгоритмами SHA-1/MD5 – см., например, [220].

Семейство алгоритмов SHA-3

В 2007 г. начался процесс пересмотра алгоритмов, лежащих в основе стандарта SHS: Национальный институт стандартов и технологий NIST объявил о проведении открытого конкурса по выбору нового стандарта хеширования США SHA-3 и опубликовал требования к алгоритмам хеширования, которые могли участвовать в данном конкурсе [156].

SHA-3 должен был прийти на смену алгоритмам хеширования семейства SHA-2. Конкурс SHA-3 подразумевал выбор наилучшего алгоритма хеширования по целому ряду критерии, основными из которых являлись криптостойкость алгоритма и его быстродействие.

В 2008 г. на конкурс SHA-3 было прислано 64 алгоритма хеширования. Основой нового стандарта хеширования стал алгоритм Кессак [71], который был выбран сообществом экспертов как наилучший из претендентов [190]. Авторы алгоритма Кессак – Гвидо Бертони, Джоан Деймен, Михаэль Петерс и Жиль Ван Аске.

Драфт стандарта, описывающего семейство алгоритмов SHA-3, основанное на алгоритме Кессак и включающее в себя алгоритмы хеширования SHA3-224, SHA3-256, SHA3-384 и SHA3-512, вышел в мае 2014 г. [132].

Структура алгоритмов

Алгоритмы семейства SHA-3 построены по принципу криптографической губки; такая структура криптографических алгоритмов была предложена авторами алгоритма Кессак ранее [69] и рассмотрена в данной главе выше.

Алгоритм семейства SHA-3 выполняется в три этапа [70, 71, 132].

Этап 1. Инициализация алгоритма. В рамках данного этапа выполняется обнуление внутреннего состояния алгоритма, дополнение входных данных и их разбиение на блоки.

Входное сообщение разбивается на блоки, размер которых зависит от размера выходного значения конкретного из алгоритмов семейства SHA-3 следующим образом:

$$R = 1600 - 2 * n,$$

где:

- R – размер блока хешируемого сообщения в битах;
- n – размер выходного значения алгоритма SHA3- n .

Константа 1600 соответствует размеру внутреннего состояния алгоритма в битах (см. далее).

Дополнение последнего блока выполняется следующим образом (как и, например, для алгоритма MD5, это может привести к появлению дополнительного блока данных):

- добавляется единичный байт (т. е. байт с шестнадцатеричным значением 01);
- добавляется единичный бит;
- добавляется при необходимости такое количество нулевых битов, чтобы текущий размер последнего блока сообщения был равен $R - 1$;
- добавляется завершающий единичный бит.

Этап 2. Этап «впитывания», который включает в себя наложение текущего блока данных на внутреннее состояние и последующее перемешивание состояния алгоритма.

Внутреннее состояние алгоритма (изначально заполненное нулевыми битами) S представляется в виде одномерного массива размером 1600 бит. В рамках данного этапа для каждого блока дополненного сообщения выполняются следующие действия в соответствии со структурой криптографической губки (см. рис. 1.2):

1. На первые R бит состояния операцией XOR накладывается блок входных данных.
2. Состояние обрабатывается функцией $f()$, которая является основой алгоритма и выполняет перемешивание его состояния.

В процессе преобразований, выполняемых функцией $f()$, состояние представляется в виде двумерного массива A размером 5×5 , элементами которого являются 64-битовые слова. Конвертация одномерного массива состояния S в массив A производится следующим образом:

$$A[x, y] = S'[5y + x],$$

где S' обозначает представление массива S в виде одномерного массива 64-битовых слов.

Функция $f()$ выполняет 24 раунда, в каждом из которых производятся следующие действия:

$$C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4], x = 0 \dots 4;$$

$$D[x] = C[x - 1] \oplus (C[x + 1] \ggg 1), x = 0 \dots 4;$$

$$A[x, y] = A[x, y] \oplus D[x], x = 0 \dots 4, y = 0 \dots 4;$$

$$B[y, 2x + 3y] = A[x, y] \ggg r[x, y], x = 0 \dots 4, y = 0 \dots 4;$$

$$A[x, y] = B[x, y] \oplus (\sim B[x + 1, y] \& B[x + 2, y]), x = 0 \dots 4, y = 0 \dots 4;$$

$$A[0, 0] = A[0, 0] \oplus RC[i],$$

где:

- B – временный массив, аналогичный по структуре массиву состояния;
- C и D – временные массивы, содержащие по 5 слов размером 64 бита каждое;
- r – массив, определяющий количество битов вращения для каждого слова состояния (см. [71]);

- \sim – побитовый комплемент;
- $>>>$ – операция побитового циклического сдвига вправо на указанное количество битов;
- $RC[i]$ – константа текущего раунда i ($i = 0 \dots 23$); константы RC приведены в приложении 1;
- операции с индексами массива выполняются по модулю 5.

Данные операции (кроме операции наложения раундовой константы) показаны на рис. 1.44, где их последовательность обозначена цифрами.

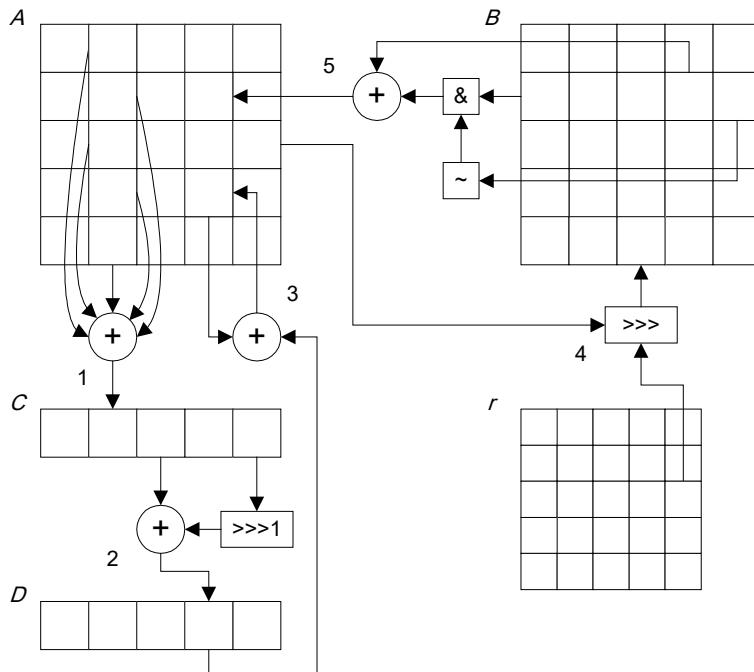


Рисунок 1.44. Схематичное изображение раунда функции $f()$ алгоритма SHA-3

Отметим, что некоторые из операций допускают различные варианты реализации (в частности, операция вращения слов состояния на r бит), из которых в зависимости от ресурсов конкретной программной или аппаратной платформы может быть выбран наиболее оптимальный. Раундовые константы также являются вычисляемыми, что позволяет в случае крайне ограниченных ресурсов не хранить таблицу констант (см. [132]).

Этап 3. «Выжимание», в процессе которого из внутреннего состояния вычисляется результирующий хеш-код алгоритма.

Хеш-код вычисляется путем нескольких вызовов функции $f()$, после каждого из которых оно дополняется некоторыми битами внутреннего состояния алгоритма. Данный этап состоит из выполнения следующих шагов:

1. Обозначим массив промежуточного выходного значения как Z ; изначально данный массив пуст, т. е. имеет нулевой размер.
2. Стока Z дополняется R первыми битами состояния.

3. Если размер Z становится больше или равен n (т. е. размеру выходного значения алгоритма), то Z усекается до требуемого размера, в результате чего получается выходное значение и работа алгоритма завершается.
4. В противном случае выполняется функция $f()$ и производится возврат к шагу 2 данного этапа.

Отметим, что для алгоритмов SHA3-224, SHA3-256, SHA3-384 и SHA3-512 справедливо соотношение $R \geq n$, поэтому этап «выжимания» сводится к однократному выполнению описанных выше шагов 1–3.

Помимо описанных выше четырех алгоритмов семейства SHA-3 с фиксированными размерами выходного значения, стандарт [132] описывает также два алгоритма хеширования с переменным размером выходного значения: SHAKE128 и SHAKE256 (где n в SHAKE n является не размером выходного значения, а определяет теоретический уровень криптостойкости алгоритма). Предполагается, что применение данных алгоритмов будет регулироваться последующими руководящими документами NIST.

Быстродействие алгоритмов

Еще в процессе проведения конкурса SHA-3 алгоритм Кессак показал стабильно высокое быстродействие на всех платформах, используемых для тестирования быстродействия криптографических алгоритмов в рамках проекта eBACS (ECRYPT Benchmarking of Cryptographic Systems – сравнительный анализ криптосистем в рамках проекта ECRYPT) [110].

Эксперты NIST в [190] назвали Кессак наиболее быстрым в аппаратной реализации по сравнению как с SHA-2, так и с другими алгоритмами-финалистами конкурса SHA-3. Быстродействие Кессак в программной реализации также посчитали приемлемым.

Кроме того, выбор Кессак в качестве основы нового стандарта хеширования предопределили наличие у него высокого запаса криптостойкости, его гибкость, позволяющая адаптировать конструкцию алгоритма под различные применения, а также простота и элегантность его структуры [95].

Отличия алгоритмов Keccak и SHA-3

По сравнению с описанным выше семейством алгоритмов SHA-3 оригинальный алгоритм Keccak [71] имеет ряд отличий, в частности:

- Keccak значительно более гибок – он имеет множество настраиваемых параметров с целью обеспечения оптимального соотношения криптостойкости и быстродействия для определенного применения алгоритма на определенной платформе; настраиваемыми величинами, в частности, являются: размер блока данных, размер состояния алгоритма, количество раундов в функции $f()$ и др.;
- в алгоритме Keccak иначе выполняется дополнение последнего блока хешируемых данных.

Использование алгоритмов семейств SHA-3 и SHA-2

После появления семейства алгоритмов SHA-3 использование предыдущих стандартов хеширования не было отменено. Как было сказано ранее, последняя на момент подготовки данной книги версия документа, описывающая стандарты хеширования SHA-1 и SHA-2, вышла в 2012 г. [129].

При этом использование алгоритма SHA-1 при вычислении электронной подписи было запрещено после декабря 2013 г., тогда как прочие применения SHA-1 и использование алгоритмов семейства SHA-2 в любых целях разрешались без ограничений по времени [60]. На текущий момент предполагается, что в последующих руководящих документах NIST будут даны рекомендации по предпочтительному применению алгоритмов SHA-2 или SHA-3 в конкретных случаях.

Результаты криптоанализа

На текущий момент не опубликовано каких-либо атак на полнораундовые версии алгоритмов семейства SHA-3.

1.3.4 Отечественные стандарты хеширования

Данный раздел посвящен описанию отечественных стандартов хеширования, которые широко используются в России. Предпринимаются также усилия по международной стандартизации текущего стандарта хеширования РФ [26].

ГОСТ Р 34.11–94

Долгое время стандартом хеширования РФ был ГОСТ Р 34.11–94 [14]. Лежащий в его основе алгоритм хеширования базируется на использовании отечественного стандарта шифрования ГОСТ 28147–89 [11] в качестве внутреннего блочного шифра.

Структура алгоритма

Алгоритм выполняется в три этапа. На первом этапе выполняется инициализация внутреннего состояния алгоритма следующим образом [14]:

$$M = M_0;$$

$$H = H_0;$$

$$Sum = 0;$$

$$L = 0,$$

где:

- M_0 – хешируемое сообщение;
- M – текущая необработанная часть хешируемого сообщения;
- H_0 – стартовый вектор хеширования;
- H – текущее 256-битовое значение хеш-функции;
- Sum – текущее значение контрольной суммы;
- L – текущее значение длины обработанной на предыдущих итерациях части входной последовательности.

256-битовый стартовый вектор хеширования явным образом в стандарте ГОСТ Р 34.11–94 не определен. Таким образом, значение H_0 является дополнительным параметром алгоритма.

На втором этапе анализируется размер в битах $|M|$ необработанной части сообщения M ; если значение $|M|$ не превышает 256, осуществляется переход к третьему этапу.

Затем необработанная часть сообщения разбивается на блоки по 256 бит путем разбиения входного сообщения на две части – M_p и M_s , одна из которых (M_s) имеет размер 256 бит:

$$M = M_p \parallel M_s.$$

После этого выполняются следующие действия с каждым из блоков M_s до тех пор, пока размер необработанной части сообщения не станет меньше или равным 256 бит:

$$H = \chi(M_s, H);$$

$$L = L + 256 \bmod 2^{256};$$

$$Sum = Sum + M_s \bmod 2^{256};$$

$$M = M_p,$$

где в операции обновления контрольной суммы строка M_s конвертируется в числовое значение (аналогичное действие выполняется и в третьем этапе), а функция $\chi()$ представляет собой основное преобразование алгоритма – шаговую функцию хеширования, которая подробно описана далее.

На третьем этапе текущая необработанная часть сообщения M дополняется слева битовыми нулями до 256 бит с получением 256-битовой строки M' ; после этого выполняются следующие действия:

$$L = L + |M| \bmod 2^{256};$$

$$Sum = Sum + M' \bmod 2^{256};$$

$$H = \chi(M', H);$$

$$H = \chi(L, H);$$

$$H = \chi(Sum, H).$$

Значение H после выполнения последнего из перечисленных выше действий является выходным 256-битовым значением алгоритма хеширования.

Шаговая функция хеширования также выполняется в несколько этапов. Входными данными функции $\chi()$ являются текущее значение хеш-функции H и обрабатываемый 256-битовый блок M_s хешируемого сообщения (или другой блок аналогичного размера на третьем этапе работы алгоритма).

На первом этапе генерируются ключи шифрования на основе входных данных. Эта процедура выполняется с помощью следующей последовательности шагов:

1. Инициализируются внутренние переменные i, U, V :

$$i = 1;$$

$$U = H;$$

$$V = M_s.$$

2. Выполняются следующие вычисления:

$$W = U \oplus V;$$

$$K_1 = P(W),$$

где:

- W – временные переменные;
 - $K_j, j = 1 \dots 4, - 256$ -битовые ключи внутреннего блочного шифра;
 - преобразование $P()$ будет описано далее.
3. Инкрементируется индекс текущего ключа:

$$i = i + 1.$$

4. Если значение i превышает 4, то работа процедуры формирования ключей завершается; в противном случае выполняется переход к следующему шагу.

5. Осуществляется модификация внутренних переменных:

$$U = A(U) \oplus C_i;$$

$$V = A(A(V));$$

$$W = U \oplus V,$$

где:

- функция $A()$ будет описана далее;
- $C_j, j = 2 \dots 4, -$ константы, определенные следующим образом:

$$C_2 = C_4 = 0^{256};$$

$$C_3 = 1^8 0^8 1^{16} 0^{24} 1^{16} 0^8 (0^8 1^8)^2 1^8 0^8 (0^8 1^8)^4 (1^8 0^8)^4;$$

в данном случае запись d^n обозначает битовую строку (подстроку) из n повторяющихся битов со значением d .

6. Вычисляется значение следующего ключа:

$$K_i = P(W).$$

7. Выполняется переход к шагу 3.

Функция $A()$ определена следующим образом:

$$A(X) = (x_1 \parallel x_2) \parallel x_4 \parallel x_3 \parallel x_2,$$

где $x_1 \dots x_4$ – 64-битовые подстроки 256-битовой строки X :

$$X = x_4 \parallel x_3 \parallel x_2 \parallel x_1.$$

Преобразование $P()$ трансформирует входную 256-битовую строку, рассматриваемую как последовательность 8-битовых слов $\xi_{32} \parallel \dots \parallel \xi_1$, в строку $\xi_{\varphi(32)} \parallel \dots \parallel \xi_{\varphi(1)}$, где:

$$\varphi(i + 1 + 4 * (k - 1)) = 8i + k, i = 0 \dots 3, k = 1 \dots 8.$$

На втором этапе выполняется зашифрование текущего значения хеш-функции H с помощью сформированных на предыдущем этапе ключей шифрования.

Для этого 256-битовое значение H рассматривается в виде четырех 64-битовых фрагментов, каждый из которых шифруется на отдельном ключе K_i , $i = 1 \dots 4$, в режиме простой замены с помощью алгоритма блочного шифрова-

ния, определенного в стандарте ГОСТ 28147–89 [11]. В результате зашифрования получается 256-битовое значение S .

Поскольку в стандарте ГОСТ 28147–89 не определен узел замены алгоритма шифрования, значение узла замены является дополнительным параметром данного алгоритма хеширования.

На третьем этапе производится перемешивание текущих значений H , Ms и S и формирование выходного значения шаговой функции хеширования.

Выходное значение функции $\chi()$ определяется следующим образом:

$$\chi(Ms, H) = \psi^{61}(H \oplus \psi(Ms \oplus \psi^{12}(S))),$$

где:

- $\psi^n()$ – n -я степень преобразования $\psi()$;
- преобразование $\psi()$ трансформирует 256-битовое входное значение, рассматриваемое в виде 16-битовых фрагментов $\eta_{16} \parallel \dots \parallel \eta_1$, в следующее выходное значение аналогичного размера:

$$\eta_1 \oplus \eta_2 \oplus \eta_3 \oplus \eta_4 \oplus \eta_{15} \oplus \eta_{16} \parallel \eta_{15} \parallel \dots \parallel \eta_2.$$

Результаты криптоанализа

В 2008 г. была опубликована работа [172], в которой были опубликованы следующие атаки на алгоритм ГОСТ Р 34.11–94:

- атака по нахождению коллизий с трудоемкостью 2^{105} операций;
- атака по поиску прообразов с трудоемкостью 2^{192} операций.

Обе атаки не имеют практической применимости.

ГОСТ Р 34.11–2012

В связи с необходимостью ввода нового стандарта хеширования с удовлетворяющим современным требованиям уровнем криптостойкости, а также с характеристиками, соответствующими новому отечественному стандарту электронной подписи ГОСТ Р 34.10–2012 (см. [13]), в августе 2012 г. старый стандарт хеширования был заменен на ГОСТ Р 34.11–2012 [15].

Однако ГОСТ Р 34.11–94 продолжал действовать до 2018 г. включительно в рамках переходного периода, в основном с целью обеспечения совместимости с разработанными ранее системами, основанными на его использовании [27].

ГОСТ Р 34.11–2012 разработан Центром защиты информации и специальной связи ФСБ России с участием ОАО «ИнфоТеКС» [15]. Альтернативное неофициальное название алгоритма, лежащего в основе стандарта ГОСТ Р 34.11–2012, – «Стрибог».

Структура алгоритма

Алгоритм обрабатывает входное сообщение блоками по 512 бит. На верхнем уровне структура алгоритма представляет собой незначительно модифицированную схему Меркля–Дамгорда [29], при этом выходное значение алгоритма может иметь 256- или 512-битовый размер.

Выполнение алгоритма может быть представлено следующими тремя этапами.

Этап 1. Инициализация внутреннего состояния алгоритма. В рамках этого этапа выполняются следующие действия:

$$h = IV;$$

$$N = \Sigma = 0,$$

где:

- h – 512-битовый регистр внутреннего состояния алгоритма;
- IV – вектор инициализации; его значение зависит от размера выходного значения алгоритма: IV заполнен битовыми нулями при вычислении 512-битового хеш-кода и заполнен 8-битовыми последовательностями «00000001» при вычислении 256-битового выходного значения;
- N – 512-битовый счетчик обработанных данных;
- Σ – 512-битовый регистр, содержащий текущую сумму обработанных данных по модулю 2^{512} .

Этап 2. Поочередная обработка каждого блока сообщения, кроме последнего блока, если он является неполным, состоящая из следующих операций:

$$h = g(h, M, N);$$

$$N = N + 512;$$

$$\Sigma = \Sigma + M,$$

где:

- M – 512-битовый блок хешируемого сообщения;
- $g()$ – функция сжатия, которая подробно будет описана далее;
- сложение выполняется по модулю 2^{512} (здесь и далее).

Этап 3. Дополнение последнего блока хешируемых данных и завершающие преобразования.

Последний блок сообщения всегда дополняется единичным битом, а также, при необходимости, битовыми нулями до достижения размера, кратного 512 битам. Дополнение увеличивает количество блоков данных на один, если размер сообщения изначально был кратен 512 битам.

После этого выполняются следующие финализирующие операции:

$$h = g(h, M, N);$$

$$N = N + (l \bmod 512);$$

$$\Sigma = \Sigma + M;$$

$$h = g(h, N, 0);$$

$$H = g(h, \Sigma, 0),$$

где l – размер хешируемого сообщения (до его дополнения).

В качестве результата работы 512-битового алгоритма ГОСТ Р 34.11–2012 берется значение H . Для 256-битового варианта алгоритма результатом являются старшие 256 бит значения H .

Функция сжатия $g()$ основана на внутреннем блочном шифре E() и построена на обсуждавшейся ранее конструкции Мияучи–Пренеля [182] (на рис. 1.45

показана схема интерпретации конструкции Миягучи–Пренеля в алгоритме ГОСТ Р 34.11–2012):

$$g(h, M, N) = E(k(h, N), M) \oplus h \oplus M,$$

где $k()$ – функция вычисления псевдоключа блочного шифра (см. далее).

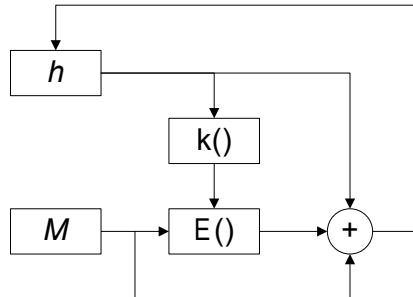


Рисунок 1.45. Конструкция Миягучи–Пренеля в алгоритме ГОСТ Р 34.11–2012

Лежащий в основе функции сжатия 512-битовый блочный шифр $E(K, M)$ выполняется в 12 раундов. M – это блок шифруемых данных, а K – псевдоключ, который обновляется в каждом раунде алгоритма.

Фактически преобразования данных в рамках блочного шифра $E()$ выполняются в два потока, в которых используется один и тот же набор преобразований:

- первый поток модифицирует псевдоключ;
- второй поток шифрует блок данных.

Структура блочного шифра $E()$ приведена на рис. 1.46. В нем используется следующая последовательность операций:

$$E(K, M) = X[K_{13}]LPSX[K_{12}] \dots LPSX[K_2]LPSX[K_1](M).$$

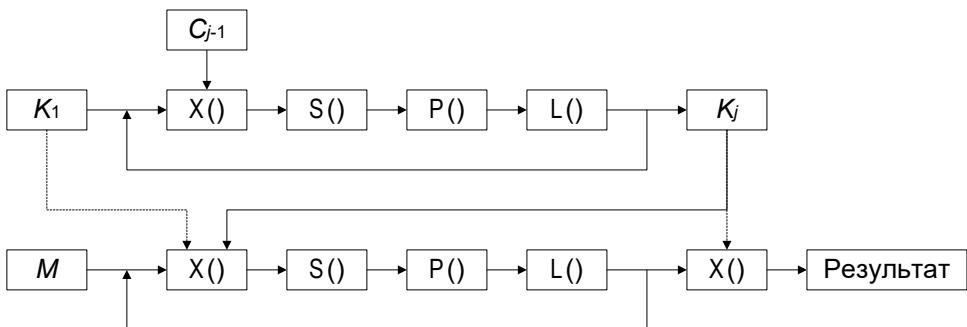


Рисунок 1.46. Структура внутреннего блочного шифра алгоритма ГОСТ Р 34.11–2012

$X[t](a)$ – операция наложения псевдоключа (вычисление и модификация псевдоключа будут подробно описаны далее) путем суммирования по модулю 2:

$$X[t](a) = t \oplus a,$$

где:

- a – текущее состояние блочного шифра, изначально $a = M$;
- в качестве t используется значение подключа текущего раунда.

$S(a)$ – побайтовая замена состояния согласно таблице замен, приведенной в приложении 1.

$P(a)$ – перестановка байтов состояния согласно таблице, приведенной в приложении 1.

$L(a)$ – умножение на матрицу A в поле GF(2). Матрица A приведена в приложении 1.

Раундовые псевдоключи K_i вычисляются следующим образом. Исходный псевдоключ K_1 зависит от текущего состояния алгоритма и счетчика N :

$$K_1 = k(h, N) = LPS(h \oplus N).$$

Дальнейшая модификация псевдоключей выполняется с использованием тех же описанных выше преобразований:

$$K_j = LPS(K_{j-1} \oplus C_{j-1}),$$

где:

- j – номер раунда;
- C_i – раундовые константы согласно таблице, приведенной в приложении 1.

Результаты криптоанализа

После появления нового российского стандарта функции хеширования привлек достаточно пристальное внимание специалистов в области защиты информации, в том числе криптоаналитиков. На текущий момент появилось несколько работ по анализу алгоритма ГОСТ Р 34.11–2012, однако пока не предложено каких-либо атак на данный алгоритм, близких к практической применимости.

Глава 2

Алгоритмы электронной подписи на эллиптических кривых

В блокчейн-технологиях широко используются алгоритмы электронной подписи – в основном для обеспечения связи между блоками цепочек данных и для обеспечения целостности данных.

Современные алгоритмы электронной подписи базируются на использовании эллиптических кривых.

Данная глава рассматривает эллиптические кривые, лежащие в основе алгоритмов электронной подписи, а также дает описание ряда широко используемых алгоритмов электронной подписи. В начале главы приведены основные математические определения, имеющие отношения к данной предметной области.

2.1 МАТЕМАТИЧЕСКИЕ ОСНОВЫ

Приведем в данном разделе определения основных математических терминов и понятий, которые будут использованы в этой главе.

Определим понятие бинарной операции для множества S . Под $S \times S$ обозначим множество всех возможных упорядоченных пар (s_1, s_2) , где s_1 и s_2 принадлежат S .

Бинарная операция на множестве S – это отображение множества $S \times S$ во множество S .

Самый простой пример бинарной операции на множестве – это множество целых чисел Z с операциями сложения и умножения, которые являются бинарными операциями на Z .

Группой называется множество G с бинарной операцией $*$, если выполняются следующие условия:

1. Ассоциативность: для любых $a, b, c \in G$:

$$a * (b * c) = (a * b) * c.$$

2. Существует нейтральный элемент $e \in G$: для любого элемента $a \in G$:

$$a * e = e * a = a.$$

3. Для любого $a \in G$ существует обратный элемент a^{-1} :

$$a * a^{-1} = a^{-1} * a = e.$$

Если также для любых $a, b \in G$ выполняется $a * b = b * a$, то группа называется абелевой, или коммутативной.

Если бинарная операция в группе обозначается как сложение, то $a * b$ заменяется на $a + b$, нейтральный элемент e – на 0, а обратный a элемент a^{-1} – на $-a$. Такая группа называется аддитивной. Группа же с операцией умножения, определенная выше, называется мультипликативной.

Определение аддитивной группы G будет выглядеть так:

1. Ассоциативность: для любых $a, b, c \in G$:

$$a + (b + c) = (a + b) + c.$$

2. Существует нейтральный элемент 0 $\in G$: для любого элемента $a \in G$:

$$a + 0 = 0 + a = a.$$

3. Для любого $a \in G$ существует обратный элемент $-a$:

$$a + -a = -a + a = 0.$$

Если также для любых $a, b \in G$ выполняется $a + b = b + a$, то группа называется абелевой, или коммутативной.

Для аддитивной группы сложение элемента a и элемента, обратного элементу b , чаще всего обозначают не как $a + -b$, а просто $a - b$.

Нейтральный элемент в мультипликативной группе называется единичным элементом, или единицей, а в аддитивной группе – нулевым элементом, или нулем.

Примеры групп:

1. Множество целых чисел Z с операцией сложения.

Очевидно, что сложение ассоциативно: для любых целых a, b, c выполняется $a + (b + c) = (a + b) + c$.

Нейтральный элемент – это 0. Для каждого целого a имеется обратный элемент $-a$.

Кроме того, сложение коммутативно, поэтому Z с операцией сложения – абелева группа.

2. Множество из двух целых чисел $\{-1, 1\}$ с операцией умножения.

Ассоциативность выполняется. Нейтральным элементом является 1. Для -1 обратным является -1 .

Операция коммутативна, поэтому это тоже абелева группа.

3. Множество классов вычетов по модулю n с операцией сложения: числа $\{0, \dots, n - 1\}$ – остатки от деления на n образуют абелеву группу с нейтральным элементом 0.

4. Множество положительных вещественных чисел с умножением.

Ассоциативность выполняется, 1 – нейтральный элемент. Для любого целого положительного a есть обратный элемент $1/a$.

Кроме того, операция коммутативна, следовательно, группа – абелева.

5. Множество классов вычетов по модулю простого числа p с операцией умножения: числа $\{1, \dots, p - 1\}$ – остатки от деления на p – образуют абелеву группу с нейтральным элементом 1.

Группы делятся на конечные и бесконечные, в зависимости от числа элементов. Например, группы № 1 и № 4 из числа приведенных выше примеров – бесконечные, а группы № 2, 3 и 5 – конечные.

Число элементов группы G называется порядком группы и обозначается как $\#G$ или $|G|$.

У групп № 1 и № 4 порядок бесконечен, у групп же № 2, 3 и 5 он равен 2, n и $p - 1$ соответственно.

Подмножество H , состоящее из элементов группы G , называют подгруппой группы G , если она сама является группой относительно групповой операции G .

Самой простой пример подгруппы – нейтральный элемент любой группы. Кроме того, чисто формально любая группа содержит саму себя и является подгруппой по отношению к самой себе. Эти две подгруппы называют тривиальными подгруппами.

Рассмотрим в качестве примера группу классов вычетов по модулю 5 с операцией умножения: $\{1, 2, 3, 4\}$. Ее порядок равен 4. Она содержит следующие подгруппы: $\{1\}$, $\{1, 4\}$, $\{1, 2, 3, 4\}$.

Теорема Лагранжа:

Порядок любой подгруппы H делит без остатка порядок содержащей ее группы G .

Эту теорему можно кратко записать таким образом:

$$\#H \mid \#G,$$

если H – подгруппа группы G .

Символ \mid в данном случае означает деление без остатка. Например, $a \mid b$ означает, что a без остатка делит b .

Порядком элемента a мультиликативной группы называется такое минимальное натуральное число k , что:

$$a^k = \underbrace{a * \dots * a}_{k \text{ раз}} = \text{нейтральный элемент.}$$

Для аддитивной группы экспонента a^k заменяется на ka :

$ka = \underbrace{a + \dots + a}_{k \text{ раз}} = \text{нейтральный элемент}$ (т. е. получается аналогичное определение порядка элемента).

Например, порядок элемента -1 из группы $\{-1, 1\}$ с операцией умножения равен двум.

Если в группе порядка n существует хотя бы один элемент порядка n , то такую группу называют циклической. Такой элемент называют образующим элементом (или генератором группы).

Рассмотренные выше группы № 2, 3 и 5 – циклические. В группе № 2 генератором является -1 , а в группе № 3 – любой взаимно простой с n элемент (т. е. не имеющий с n общих делителей). Например, в группе № 3 генератором является

число 1, поскольку, складывая 1, можно получить все элементы группы, начиная с 1 и заканчивая нейтральным элементом 0.

В группе № 5 генераторами являются элементы 2 и 3. Действительно, вся группа № 5 – это множество $\{1, 2, 3, 4\}$ с операцией умножения mod 5. Для степеней 2 имеем:

$$2^1 \text{ mod } 5 = 2;$$

$$2^2 \text{ mod } 5 = 4;$$

$$2^3 \text{ mod } 5 = 3;$$

$$2^4 \text{ mod } 5 = 1.$$

То есть степени элемента 2 «генерируют» всю группу. Порядок элемента 2 равен порядку группы.

Аналогично ведет себя элемент 3:

$$3^1 \text{ mod } 5 = 3;$$

$$3^2 \text{ mod } 5 = 4;$$

$$3^3 \text{ mod } 5 = 2;$$

$$3^4 \text{ mod } 5 = 1.$$

А степени 4 образуют подгруппу порядка 2:

$$4^1 \text{ mod } 5 = 4;$$

$$4^2 \text{ mod } 5 = 1.$$

Таким образом, 4 – генератор подгруппы $\{1, 4\}$ группы $\{1, 2, 3, 4\}$, и его порядок равен двум. Также стоит отметить, что порядок подгруппы $\{1, 4\}$ равен двум, что является делителем числа четыре (порядка группы $\{1, 2, 3, 4\}$) в соответствии с теоремой Лагранжа.

Кольцом называют множество R с двумя бинарными операциями $*$ и $+$, такими, что:

- 1) множество R с операцией $+$ представляет собой коммутативную группу;
- 2) операция $*$ ассоциативна: для любых $a, b, c \in R$

$$a * (b * c) = (a * b) * c;$$

- 3) для любых $a, b, c \in R$:

$$a * (b + c) = a * b + a * c \text{ и } (b + c) * a = b * a + c * a$$

(это свойство называют дистрибутивностью).

Кольцо называют коммутативным, если операция умножения $*$ является коммутативной.

Самым простым примером кольца является множество целых чисел.

В качестве примера конечного кольца можно взять Z/Z_n – множество классов вычетов по модулю n с операциями сложения и умножения.

Полем называется коммутативное кольцо, в котором все элементы, кроме 0 (т. е. нейтрального элемента по сложению), имеют обратные элементы для операции умножения.

Множества рациональных, вещественных и комплексных чисел являются полями.

Характеристикой поля называется порядок его единичного элемента, т. е. минимальное натуральное число p , такое, что:

$$\underbrace{1 + \dots + 1}_{p \text{ раз}} = 0.$$

Характеристика конечного поля всегда является простым числом.

Поле называется конечным (или полем Галуа), если оно состоит из конечно-го числа элементов. Число элементов конечного поля всегда равно степени его характеристики.

Чаще всего для конечного поля из p^n элементов, где p – характеристика, ис-пользуют следующее обозначение:

$$GF(p^n),$$

где GF означает Galois Field – поле Галуа, а p^n – число элементов поля.

Нередко вместо $GF(p^n)$ используют обозначение F_{p^n} .

Самый элементарный пример конечного поля – поле, состоящее из p эле-ментов, где p – простое число. Такое поле называется простым полем (prime field) и обозначается как $GF(p)$ или F_p .

В качестве важного для криптографии примера конечного поля можно взять Z/Z_p – множество классов вычетов по модулю простого числа p с операциями сложения и умножения.

Мультипликативная группа поля F_{p^n} обозначается как $F_{p^n}^*$ и состоит из всех элементов поля, за исключением 0, т. е. нейтрального элемента по отношению к операции сложения. $|F_{p^n}^*| = p^n - 1$ (порядок $F_{p^n}^*$ равен $p^n - 1$).

Теорема:

Мультипликативная группа $F_{p^n}^*$ любого конечного поля F_{p^n} – ци-клическая.

Пусть G – группа с операцией \circ , а G' – группа с операцией \circ' . Отображение $\varphi: G \rightarrow G'$, переводящее элементы G в элементы G' , называется гомоморфизмом G в G' , если для любых $a, b \in G$ выполняется:

$$\varphi(a \circ b) = \varphi(a) \circ' \varphi(b).$$

Если отображение φ еще и взаимно однозначно (т. е. любому элементу $\varphi(a)$ из G' соответствует один-единственный $a \in G$), то оно называется изоморфизмом.

2.2 ЭЛЛИПТИЧЕСКИЕ КРИВЫЕ

Идею использовать эллиптические кривые для построения криптографиче-ских схем предложили в 1985 г. Виктор Миллер (Victor Miller) и Нил Коблиц (Neal Koblitz) независимо друг от друга.

2.2.1 Определение эллиптической кривой

Эллиптической кривой над полем F называется множество точек (x, y) – реше-ний уравнения Вейерштрасса:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (2.1)$$

совместно с так называемой нулевой (бесконечно удаленной) точкой O (point at infinity).

Константы a_1, a_2, a_3, a_4, a_6 – это элементы поля F .

Выражение (2.1) представляет собой уравнение Вейерштрасса в общем виде (generalized Weierstrass equation) в аффинных координатах. Если характеристика поля F больше 3, то уравнение (2.1) может быть элементарно преобразовано в короткую форму (short):

$$y^2 = x^3 + ax + b. \quad (2.2)$$

Кривая должна иметь дискриминант $\Delta = 4a^3 + 27b^2$, не равный 0. При $\Delta = 0$ кривая называется сингулярной (не путайте с суперсингулярной – суперсингулярные кривые описаны далее) и не может быть использована в криптографических целях.

На практике кривые над полем с характеристикой, равной 2 (binary field), используются редко, несмотря на то что они все же присутствуют в ряде зарубежных стандартов. Для кривых над полями характеристики 3 существуют лишь теоретические работы. Поэтому далее в нашей книге мы будем рассматривать только кривые над полем с характеристикой, большей 3.

Кривая E над полем K чаще всего обозначается как $E(K)$. В криптографических алгоритмах используют кривые над конечными полями. Кривая E над конечным полем F_{p^n} (т. е. полем характеристики p , состоящем из p^n элементов) обозначается как $E(F_{p^n})$ или $E(GF(p^n))$ (данные обозначения эквивалентны, использование конкретного из них фактически зависит от того, как больше нравится авторам статей и книг обозначать конечное поле из p^n элементов).

2.2.2 Основные операции над точками эллиптической кривой

Сложение точек

Опишем групповой закон сложения точек в форме Вейерштрасса.

Точки эллиптической кривой образуют коммутативную (абелеву) группу по операции сложения, причем нулевая точка O играет роль нейтрального элемента группы: для любой точки Q имеет место равенство $Q + O = O + Q = Q$. В представлении Вейерштрасса нулевая точка O не имеет каких-либо конкретных координат (x, y) .

Пусть имеются две точки $Q_1 = (x_1, y_1)$ и $Q_2 = (x_2, y_2)$ кривой (2.2). Они считаются равными, если равны их координаты, т. е. $Q_1 = Q_2$ если $x_1 = x_2$ и $y_1 = y_2$.

Их сумма, $Q_1 + Q_2$ точка $Q_3 = (x_3, y_3)$, вычисляется следующим образом:

1. В случае, когда $Q_1 \neq Q_2$ (сложение точек):

- если $x_1 = x_2$ и $y_1 \neq y_2$, то $Q_3 = O$ (поскольку Q_1 обратна Q_2 : $Q_1 = -Q_2$);
- если $x_1 \neq x_2$, то:

$$x_3 = \lambda^2 - x_1 - x_2;$$

$$y_3 = \lambda(x_1 - x_3) - y_1,$$

$$\text{где } \lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

2. В случае, когда $Q_1 = Q_2$ (удвоение точки):

- если $y_1 = 0$, то $Q_3 = O$ (поскольку в этом случае Q_1 обратна самой себе: $Q_1 + Q_2 = 2Q_1 = O$);
- если $y_1 \neq 0$, то:

$$x_3 = \lambda^2 - 2x_1;$$

$$y_3 = \lambda(x_1 - x_3) - y_1,$$

$$\text{где } \lambda = \frac{3x_1^2 + a}{2y_1}.$$

Геометрически сумму точек можно представить с помощью двух прямых, пересекающих эллиптическую кривую, что показано на рис. 2.1 на примере кривой $y^2 = x^3 - 5x + 2$ над полем действительных чисел R .

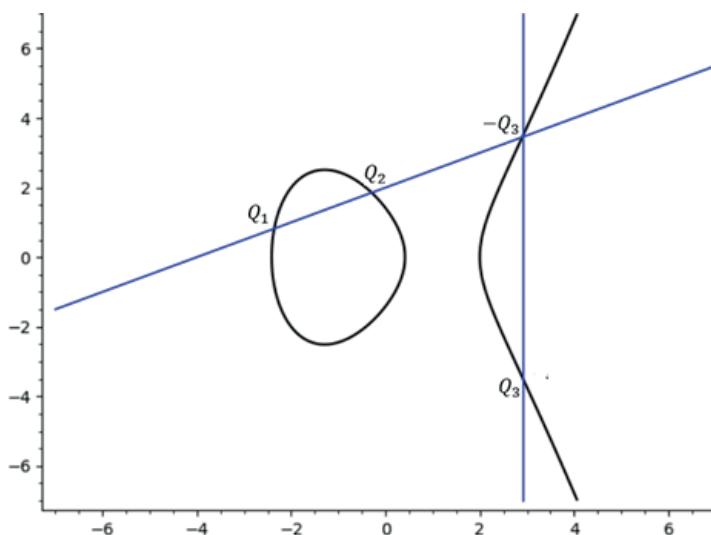


Рисунок 2.1. Геометрическое представление суммы точек эллиптической кривой

Первая прямая, проходящая через складываемые точки Q_1 и Q_2 , пересекает кривую в третьей точке $-Q_3$, обратной их сумме. Для того чтобы получить Q_3 из $-Q_3$, достаточно поменять знак у-координаты точки $-Q_3$, что геометрически выглядит как пересечение вертикальной прямой, проходящей через $-Q_3$, и кривой.

Когда складываемые точки обратны друг другу, прямая, которая проходит через них, вертикальна и не пересекает кривую больше ни в одном месте. Этот случай соответствует бесконечно удаленной точке.

Удвоение точки можно представить аналогично (рис. 2.2): через точку Q проводим касательную и, найдя ее пересечение с кривой, получаем точку $-2Q$, т. е. обратную точку $2Q$. Осталось найти требуемую точку $2Q$ при помощи вертикальной прямой, как и в случае со сложением.

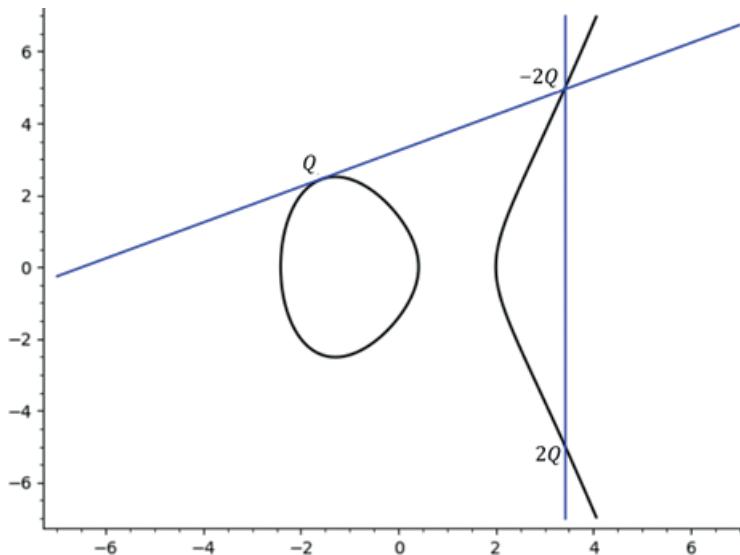


Рисунок 2.2. Геометрическое представление удвоения точки эллиптической кривой

Когда удваиваемая точка обратна самой себе (т. е. координата $y = 0$), касательная вертикальна и не пересекает кривую, что соответствует бесконечно удаленной точке.

Скалярное умножение

Скалярным умножением (scalar multiplication) числа k на точку P называется k -кратное сложение точки P :

$$Q = \underbrace{P + \dots + P}_{k \text{ раз}} = \lfloor k \rfloor P = k * P.$$

Результат этой операции (точка Q) называется точкой кратности k .

Рассмотрим, как вычисляется результат скалярного умножения. Пусть мы хотим умножить число k на точку P , т. е. узнать координаты следующей точки:

$$\lfloor k \rfloor P = \underbrace{P + \dots + P}_{k \text{ раз}}$$

Даже если число k небольшое, то вычислять $k * P$, складывая k раз точку P , было бы крайне неэффективно. Самый простой алгоритм для вычисления $k * P$ называется бинарным. Данный алгоритм предусматривает выполнение скалярного умножения следующим образом.

Сначала представим k в бинарном виде. Пусть k – это m -битовое число:

$$k = \sum_{i=0}^{m-1} k_i * 2^i,$$

где каждый из коэффициентов k_i равен 0 либо 1.

Тогда $k * P$ тоже представим в бинарном виде:

$$k * P = \sum_{i=0}^{m-1} k_i * 2^i * P.$$

Из формулы очевидно, что можно последовательно, начиная с $i = 0$ до $m - 1$, считать $2^i * P$ и прибавлять результат к общей сумме, если k_i равно 1.

Псевдокод бинарного алгоритма выглядит так (результат вычислений будет сохранен в переменной R):

```
R = 0
D = P
for i in range(m):
    if k[i] == 1:
        R = R + D
    D = 2 * D
```

Таким образом, при умножении случайного m -битового числа на точку будет выполнено m удвоений и в среднем $m/2$ сложений. Это значит, что бинарный алгоритм позволяет умножать точки на огромные числа.

К примеру, для кривой secp256k1 (будет рассмотрена далее), где для скалярного умножения используются 256-битовые числа, потребуется всего лишь 256 удвоений и около 128 сложений.

У данного алгоритма существует масса более быстрых алгоритмов-«конкурентов», которые можно использовать при практической реализации, но они выглядят более сложно и не так интуитивно понятны, как бинарный метод.

2.2.3 Основные характеристики эллиптической кривой

Параметры и характеристики кривой

Опишем характеристики, определяющие основные свойства эллиптических кривых.

Порядком группы точек кривой называется количество точек, лежащих на этой кривой (т. е. решений уравнения кривой) плюс нулевая точка.

Порядком точки P называется минимальное число k , при умножении на которое получается нулевая точка, т.е. $[k]P = O$.

Как и любая другая группа, группа точек эллиптической кривой имеет подгруппы. Если порядок кривой – простое число, то их всего две и эти подгруппы тривиальны – это нулевая точка и сама группа точек простого порядка. Как правило, в криптосистемах используют только точки из подгруппы большого простого порядка.

Количество точек кривой E над конечным полем, содержащим p^n элементов, часто в литературе обозначается как $\#E(F_{p^n})$ или $\#E(GF_{p^n})$. Существует одно весьма полезное представление этой величины:

$$\#E(F_{p^n}) = p^n + 1 - t,$$

где целое число t называется следом Фробениуса (Frobenius trace). Отметим, что оно целое, т. е. может быть нулевым, положительным или отрицательным.

Известна теорема Хассе (Hasse's theorem on elliptic curves):

След Фробениуса по абсолютной величине $\leq 2 * \sqrt{p^n}$.

Из теоремы следует, что количество точек эллиптической кривой над полем, содержащим p^n элементов, всегда лежит в относительно узком по сравнению с p^n интервале:

$$p^n + 1 - 2 * \sqrt{p^n} \leq \#E(F_{p^n}) \leq p^n + 1 + 2 * \sqrt{p^n}.$$

Это неравенство может быть полезно для грубой оценки количества точек. Порядок группы точек кривой $\#E$ чаще всего имеет следующий вид:

$$\#E = \text{cofactor} * q,$$

где q – порядок простой подгруппы (prime subgroup order) – большое простое число, а кофактор (cofactor) – как правило, небольшое целое число. q – это порядок именно той подгруппы, точки которой будут использоваться в криптографических алгоритмах.

Для практических криптографических приложений кривая описывается при помощи набора параметров – криптонабора (paramset), состоящего из следующих параметров:

- 1) конечное поле F_{p^n} (т. е. значения p и n);
- 2) коэффициенты кривой a, b (для уравнения в форме Вейерштрасса);
- 3) порядок простой подгруппы (prime subgroup order) – большое простое число q ;
- 4) кофактор (cofactor);
- 5) базовая точка P (base point) – точка порядка q (точка P – генератор подгруппы, состоящей из q элементов; иными словами, q – это минимальное натуральное число, на которое надо умножить точку P , чтобы получить бесконечно удаленную точку: $[q]P = O$).

Есть еще один важный параметр, который определяет внутреннюю структуру группы точек кривой. Он называется j -инвариантом и вычисляется по формуле:

$$j(E) = 1728 \frac{4a^3}{4a^3 + 27b^2},$$

где a и b – коэффициенты кривой E .

Если две кривые над одним и тем же полем изоморфны, то они имеют одинаковый j -инвариант. Обратное верно не всегда: у двух кривых над одним полем может быть одинаковый j -инвариант, притом что они не изоморфны.

Пусть есть эллиптическая кривая $E: y^2 = x^3 + ax + b$ над полем F . Если мы выберем некоторое значение μ , не равное 0, из этого поля и вычислим $a' = a\mu^4$ и $b' = b\mu^6$, то получим изоморфную E кривую E' : $y^2 = x^3 + a'x + b'$.

Каждой точке (x, y) на E соответствует точка $(x\mu^2, y\mu^3)$ на кривой E' . Очевидно, что после подстановки $(x\mu^2, y\mu^3)$ в уравнение E' произойдет сокращение μ^6 и мы получим уравнение $y^2 = x^3 + ax + b$. Также очевидно, что:

$$j(E') = 1728 \frac{4a^3}{4a^3 + 27b^2} = 1728 \frac{4(a\mu^4)^3}{4(a\mu^4)^3 + 27(b\mu^6)^2} = 1728 \frac{4a^3}{4a^3 + 27b^2} = j(E).$$

Изоморфизм по определению сохраняет групповой закон сложения для элементов: если на E точка (x_3, y_3) равна сумме точек (x_1, y_1) и (x_2, y_2) , то сумма образов (x_1, y_1) и (x_2, y_2) на E' равна образу (x_3, y_3) :

○ на E :

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3);$$

○ сумма образов на E' :

$$(\mu^2 x_1, \mu^3 y_1) + (\mu^2 x_2, \mu^3 y_2) = (\mu^2 x_3, \mu^3 y_3).$$

Подсчет количества точек кривой

Первый алгоритм подсчета количества точек, более эффективный, чем простой перебор, был разработан Рене Шуфом (René Schoof) в 1985 г., но он оказался неоптимальным для криптографических применений. В 1990-х годах на его основе был создан гораздо более эффективный алгоритм SEA (Schoof-Elkies-Atkin), который позволяет весьма эффективно (за время порядка $O(\log^4 p)$) решать эту задачу и реализован в ряде популярных систем компьютерной алгебры и библиотек.

2.2.4 Примеры эллиптических кривых

Проиллюстрируем сказанное выше на примерах эллиптических кривых.

Пример 1

Кривая E : $y^2 = x^3 + x + 1$ над конечным полем $F_p = F_{19}$ (или $GF(19)$ в другом обозначении).

Давайте немного поисследуем ее. Для начала вычислим значение $\#E$ (порядок группы точек).

Для «игрушечных» с точки зрения эллиптической криптографии полей, вроде F_{19} (т. е. таких полей, число элементов которых можно легко перебрать), вычисление порядка группы точек кривой проще всего провести простым перебором.

Итак, коэффициенты $a = 1$ и $b = 1$. Конечное поле F_{19} содержит простое число элементов – так называемое простое поле (prime field). Дискриминант $\Delta = 4a^3 + 27b^2 = 4 + 27 \pmod{19} = 31 \pmod{19} = 12$. $\Delta \neq 0$, следовательно, это – эллиптическая кривая (а не сингулярная) и на ней выполняются групповые операции сложения точек.

Давайте сначала выясним, какие точки она содержит. Очевидно, что для этого надо перебрать возможные значения x из поля кривой и вычислить для каждого правую часть уравнения кривой $right = x^3 + ax + b$. Если $right$ является квадратичным вычетом, то решаем уравнение $y^2 = right$ и получаем пару точек с одним x и противоположными y (т. е. (x, y) и $(x, -y)$).

Итак, для $x = 0$ значение $x^3 + x + 1$ равно 1. Символ Лежандра $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$ равен $\left(\frac{1}{19}\right) = 1$, из чего следует, что 1 – квадратичный вычет по модулю 19.

Это означает, что уравнение $y^2 = 1 \pmod{19}$ имеет решения. Так как $p = 3 \pmod{4}$, то решения для $y^2 = right \pmod{p}$ находятся по следующим формулам:

$$y_1 = a^{\frac{p+1}{4}} \pmod{p};$$

$$y_2 = -y_1 = p - y_1.$$

Поэтому $y_1 = 1$, $y_2 = -1 = 19 - 1 = 18$, следовательно, для $x = 0$ имеем два y : 1 и 18, которые удовлетворяют уравнению. Таким образом, мы получили две точки кривой: $(0, 1)$ и $(0, 18)$.

Проведя такие вычисления для всех x из поля F_{19} и отбросив повторы правой части, мы получим все решения уравнения кривой:

$$\begin{aligned} &(0, 1), (0, 18), (2, 7), (2, 12), (5, 6), (5, 13), (7, 3), \\ &(7, 16), (9, 6), (9, 13), (10, 2), (10, 17), (13, 8), (13, 11), \\ &(14, 2), (14, 17), (15, 3), (15, 16), (16, 3), (16, 16). \end{aligned}$$

Добавив к полученному множеству бесконечно удаленную точку O , получим всю группу точек кривой E . Ее порядок $\#E(F_{19})$ равен 21:

$$\#E(F_{19}) = 19 + 1 - t = 21.$$

То есть след Фробениуса $t = -1$ (как и должно быть по теореме Хассе: $|-1| \leq 2\sqrt{19}$).

Проиллюстрируем сложение точек рассматриваемой кривой.

Возьмем точки $Q_1 = (10, 2)$ и $Q_2 = (10, 17)$. Чему равна их сумма $Q_1 + Q_2$?

Данные точки обратны друг другу: $Q_1 = -Q_2$, т. к. $x_1 = x_2 = 10$ и $y_1 = -y_2$ (последнее верно, поскольку $2 = -17 \pmod{19}$). В результате $Q_1 + Q_2 = O$, так как точки являются взаимообратными.

Если же мы рассмотрим две какие-нибудь не обратные друг другу точки, например $Q_1 = (10, 17)$ и $Q_2 = (15, 3)$, то можем применить формулу сложения:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{3 - 17}{15 - 10} \pmod{19} = 1;$$

$$x_3 = \lambda^2 - x_1 - x_2 = 1 - 10 - 15 \pmod{19} = 14;$$

$$y_3 = \lambda(x_1 - x_3) - y_1 = 1 * (10 - 14) - 17 = 17.$$

Следовательно:

$$Q_3 = Q_1 + Q_2 = (14, 17).$$

Рассмотрим также удвоение точки – для $Q = (16, 3)$ вычислим $2Q$:

$$\lambda = \frac{3x_1^2 + A}{2y_1} = \frac{3 * 16^2 + 1}{2 * 3} \pmod{19} = 11;$$

$$x_3 = \lambda^2 - 2x_1 = 11^2 - 2 * 16 \pmod{19} = 13;$$

$$y_3 = \lambda(x_1 - x_3) - y_1 = 11 * (16 - 13) - 3 \pmod{19} = 11.$$

В результате $2 * Q = (13, 11)$.

Группа точек рассматриваемой кривой E – циклическая группа, т. е. в ней существуют порождающие элементы (генераторы), при помощи скалярного умножения на которые чисел от 1 до порядка группы можно получить все элементы группы.

Одна из таких точек – точка $(0, 18)$. Умножая ее на числа от 1 до 21, мы получаем все точки кривой, включая O :

$$1 * (0, 18) = (0, 18);$$

$$2 * (0, 18) = (0, 18) + (0, 18) = (5, 13);$$

$$3 * (0, 18) = (5, 13) + (0, 18) = (15, 16);$$

$$4 * (0, 18) = (15, 16) + (0, 18) = (9, 6);$$

...;

$$21 * (0, 18) = (0, 1) + (0, 18) = O.$$

Кроме того, присутствуют еще две циклические подгруппы порядков 3 и 7 (по теореме Лагранжа порядками подгрупп могут быть только делители порядка основной группы):

$$\{O, (2, 7), (2, 12)\}$$

и

$$\{O, (10, 2), (10, 17), (14, 2), (14, 17), (15, 3), (15, 16)\}.$$

Пример 2

Кривая secp256k1, которая используется в подписи ECDSA транзакций в системах Биткойн и Эфириум: $E: y^2 = x^3 + 7$ над простым полем F_p , где характеристика поля $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

В шестнадцатеричном представлении p имеет следующий вид:

$$\begin{aligned} p = & \text{FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF} \\ & \text{FFFF FFFF FFFF FFFF FFFE FFFF FC2F.} \end{aligned}$$

Коэффициенты уравнения кривой:

$$a = 0;$$

$$b = 7.$$

Число всех точек на кривой secp256k1 (также в шестнадцатеричном виде):

$$\begin{aligned} & \text{FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE} \\ & \text{BAAE DCE6 AF48 A03B BFD2 5E8C D036 4141.} \end{aligned}$$

Это простое число, из чего следует, что кофактор равен единице, т. е. $\#E = q$.

Группа точек этой кривой, в отличие от кривой из предыдущего примера, по теореме Лагранжа не содержит нетривиальных подгрупп.

Для secp256k1 базовая точка (генератор подгруппы) имеет следующие координаты в шестнадцатеричном представлении:

$x = 79BE\ 667E\ F9DC\ BBAC\ 55A0\ 6295\ CE87\ 0B07\ 029B\ FCDB\ 2DCE\ 28D9\ 59F2\ 815B\ 16F8\ 1798;$
 $y = 483A\ DA77\ 26A3\ C465\ 5DA4\ FBFC\ 0E11\ 08A8\ FD17\ B448\ A685\ 5419\ 9C47\ D08F\ FB10\ D4B8.$

Легко видеть, что число точек кривой $secp256k1$ (по теореме Хассе это порядка 2^{256}), как и любой другой кривой, которая используется в криптографических алгоритмах, невозможно посчитать простым перебором, как в первом примере.

Для этого на практике используют алгоритм SEA (Schoof-Elkies-Atkin), который позволяет это делать за считанные секунды на обычном домашнем компьютере, но, к сожалению, использует слишком сложную для этой книги математику. Поэтому мы приводим лишь пример кода на SAGE, который его использует «под капотом» и считает порядок кривой и порядок базовой точки:

```

p = 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
a = 0
b = 7
x = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
y = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
E = EllipticCurve(GF(p), [a, b])
BasePoint = E(x, y)
pointOrder = BasePoint.order()
curveGroupOrder = E.order()
print("Order of group of points = " + hex(curveGroupOrder))
print("Order of BasePoint = " + hex(pointOrder))

```

Пример 3

Кривая bn254, применяемая во многих протоколах, основанных на спариваниях.

$E: y^2 = x^3 + 2$ над простым полем F_p .

Основные параметры кривой bn254:

- характеристика поля:

$$\begin{aligned}
 p &= 2523\ 6482\ 4000\ 0001\ BA34\ 4D80\ 0000\ 0008 \\
 &\quad 6121\ 0000\ 0000\ 0013\ A700\ 0000\ 0000\ 0013;
 \end{aligned}$$

- j -инвариант = 0 (т. к. $a = 0$);

- порядок простой подгруппы:

$$\begin{aligned}
 q &= 2523\ 6482\ 4000\ 0001\ BA34\ 4D80\ 0000\ 0007 \\
 &\quad FF9F\ 8000\ 0000\ 0010\ A100\ 0000\ 0000\ 000D;
 \end{aligned}$$

- количество всех точек кривой: $\#E = q$ – простое, т. е. кофактор = 1;
- базовая точка:

$$\begin{aligned}
 x &= 2523\ 6482\ 4000\ 0001\ BA34\ 4D80\ 0000\ 0008 \\
 &\quad 6121\ 0000\ 0000\ 0013\ A700\ 0000\ 0000\ 0012; \\
 y &= 1.
 \end{aligned}$$

Характеристика поля, порядок простой подгруппы и x -координата базовой точки приведены в шестнадцатеричном представлении.

2.2.5 Задача дискретного логарифмирования в группе точек эллиптической кривой

Опишем задачу дискретного логарифмирования в группе точек эллиптической кривой, на сложности решения которой основываются криптографические алгоритмы на эллиптических кривых.

Описание задачи

Как было показано ранее, задача умножения точки на число решается быстро даже для больших чисел.

Теперь рассмотрим обратную задачу. К примеру, мы уже посчитали $Q = [k]P$. Насколько будет сложно вычислить число k , зная только лишь точки P и Q ?

Решение такой задачи называется дискретным логарифмированием в группе точек эллиптической кривой. Задача нахождения дискретного логарифма в группе точек эллиптической кривой (Elliptic Curve Discrete Logarithm Problem, ECDLP) является вычислительно сложной, так же, как и задачи факторизации (Factoring) и дискретного логарифмирования в мультиплексивной группе конечного поля (Discrete Logarithm Problem, DLP), но, в отличие от последних, имеет экспоненциальную сложность решения (напомним, что для задач Factoring и DLP существуют субэкспоненциальные алгоритмы). Как следствие криптосистемы, построенные на эллиптических кривых, более эффективны – они имеют гораздо более короткие длины ключей и работают быстрее.

Самый эффективный на данный момент метод для решения ECDLP называется ро-методом Полларда (Pollard's ρ -method). Данный метод был изначально разработан Джоном Поллардом в 1978 г. для решения DLP, но после появления эллиптических кривых в криптографии его стали использовать и для решения ECDLP.

Этот вероятностный алгоритм имеет сложность, зависящую от порядка q подгруппы, которой принадлежат точки P и Q . Он требует порядка \sqrt{q} операций с точками.

То есть, к примеру, для решения ECDLP на рассмотренной ранее кривой secp256k1, которая имеет порядок простой подгруппы q приблизительно 2^{256} , сложность «взлома» будет $\sim 2^{128}$, что невозможно на существующем этапе развития вычислительной техники.

Таким образом, стоимость (трудоемкость) решения ECDLP ро-методом Полларда составляет приблизительно \sqrt{q} операций.

Атака методом Поллига–Хеллмана

Отметим, что оценка уровня безопасности кривой при помощи \sqrt{q} применима только для случая, когда q – простое число, т. е. точки P и Q лежат именно в подгруппе простого, а не составного порядка. Для групп составного порядка существует атака методом Поллига–Хеллмана (Pohlig-Hellman), которая сводит решение ECDLP в группе составного порядка к решению нескольких ECDLP в подгруппах простого порядка этой группы.

Например, если q – не простое число, а произведение нескольких простых чисел ($q = q_1 * \dots * q_n$), то стоимость решения ECDLP оценивается как сумма стои-

мостей решения ECDLP в подгруппах порядка q_1, \dots, q_n и может оказаться практически возможным решить ECDLP для данной группы, если получится решить ECDLP для каждой из n подгрупп (например, методом Полларда).

Очевидно, что использовать в крипtosистемах такую подгруппу составного порядка было бы бессмысленно с точки зрения соотношения производительности и безопасности.

Кроме того, есть пара семейств кривых, когда метод Полларда требует большого количества вычислений, но атакующий может применить гораздо более эффективные методы для вычисления дискретного логарифма:

- аномальные кривые (anomalous curves);
- кривые с малой степенью вложения (embedded degree).

Аномальные кривые

Аномальные кривые – это кривые, число точек которых равно числу элементов поля, т. е. $\#E(F_{p^n}) = p^n$.

Быстрый полиномиальный алгоритм для решения ECDLP для аномальных кривых был независимо открыт рядом криптографов в середине 1990-х годов.

Кривые с малой степенью вложения

Степенью вложения для кривой $E(F_{p^n})$ с большой простой подгруппой порядка q называется минимальное натуральное число c , такое, что q делит без остатка $p^{c*n} - 1$.

Что это значит? $p^{c*n} - 1$ – это число элементов в мультиликативной группе поля $F_{p^{c*n}}$ (из поля $F_{p^{c*n}}$, которое содержит p^{c*n} элементов, выбрасываем 0 и получаем $p^{c*n} - 1$ элементов – мультиликативную группу поля $F_{p^{c*n}}$; эта группа обозначается при помощи звездочки: $F_{p^{c*n}}^*$).

Существование делителя q у числа $p^{c*n} - 1$ (т. е. у порядка группы $F_{p^{c*n}}^*$) говорит нам о том, что в ней существует подгруппа порядка q . Известная атака MOV (Menezes-Okamoto-Vanstone) использует спаривания Вейля (Weil pairing), о которых будет рассказано далее, билинейно отображает точки кривой в элементы $F_{p^{c*n}}^*$ и, таким образом, сводит задачу ECDLP к решению задачи DLP в $F_{p^{c*n}}^*$.

Таким образом, сложность атаки определяется наиболее эффективным методом решения DLP. На настоящий момент таковым является метод решета числового поля NFS (Number Field Sieve).

К кривым с малой степенью вложения, в частности, относятся так называемые суперсингулярные кривые (кривые, у которых след Фробениуса равен $0 \bmod p$). Для любой суперсингулярной кривой степень вложения ≤ 6 . Чем это плохо? Поскольку сложность NFS существенно зависит от числа элементов поля, то чем меньше c , тем проще считать дискретный логарифм.

Для кривых над простым полем характеристики p из теоремы Hasse следует, что след Фробениуса t по абсолютной величине должен быть $\leq 2\sqrt{p}$. Поэтому для суперсингулярных кривых над простым полем он может быть равен только 0, из чего следует, что для них число точек равно $\#E(F_p) = p + 1$.

Очевидно, что самым легким случаем для атакующего была бы степень вложения, равная 1. Это может быть при $t = 2$: $\#E(F_{p^n}) = p^n + 1 - t = p^n + 1 - 2 = p^n - 1$ (поскольку q (порядок большой простой подгруппы) всегда делит $\#E(F_{p^n})$, порядок всей группы точек).

Когда можно использовать такие кривые? Очевидно, что начиная с определенного значения степени вложения метод NFS потребует выполнения такого же числа операций, как и ро-метод Полларда. В настоящий момент считается безопасным использовать на практике степень вложения, равную 12. Примером кривой с $c = 12$ является рассмотренная ранее кривая bn254.

Стоит отметить, что благодаря последним достижениям в развитии метода NFS на самом деле теоретическая стоимость MOV-атаки для кривой bn254 стала существенно меньше (2^{100}), чем стоимость ро-метода Полларда ($2^{127} = \sqrt{q}$ операций).

2.2.6 Альтернативные формы представления эллиптических кривых

Ранее была описана каноническая форма представления эллиптических кривых, называемая формой Вейерштрасса. Рассмотрим альтернативные формы представления эллиптических кривых, которые могут быть полезны в ряде применений.

Скрученные кривые Эдвардса

Скрученная кривая Эдвардса (twisted Edwards curve) над полем F_{p^n} , где $p > 3$, имеет следующую форму:

$$ex^2 + y^2 = 1 + dx^2y^2,$$

где коэффициенты e и d – такие, что $ed(e - d) \neq 0$.

То есть коэффициенты e и d – оба ненулевые и имеют разные значения. Это условие несингулярности кривой, аналогичное условию $4a^3 + 27b^2 \neq 0$ для формы Вейерштрасса. Из его выполнения следует, что групповой закон будет выполняться для всех ее точек (кроме того, сингулярные кривые не считаются эллиптическими).

Групповой закон сложения точек для кривых в данной форме выглядит так:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + x_2 y_1}{1 + x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - e x_1 x_2}{1 - x_1 x_2 y_1 y_2} \right).$$

Нулевая точка имеет координаты, равные $(0, 1)$, в отличие от формы Вейерштрасса, где она не имеет координат в принципе.

Элемент группы точек, обратный элементу (x, y) , равен $(-x, y)$, т. е.

$$(x, y) + (-x, y) = (0, 1).$$

j -инвариант $E_{e,d}^E$ скрученной кривой Эдвардса с коэффициентами e, d :

$$j(E_{e,d}^E) = \frac{16(e^2 + 14ed + d^2)^3}{ed(e-d)^4}.$$

Легко видеть самое важное отличие от группового закона для уравнения Вейерштрасса: для формы Эдвардса используется одна и та же формула как для сложения, так и для удвоения точек.

Это существенно упрощает ее практическую реализацию и усложняет так называемые атаки по побочным каналам (side-channel attack), когда противник, имея доступ к устройству, производящему вычисления, может воспользоваться разницей в мощности потребления (или излучения) энергии процессором при удвоении и сложении точек и узнать ключевую информацию (или, по крайней мере, предположить с высокой долей вероятности).

Самый простой практический пример атаки по побочным каналам – это SPA (Simple Power Analysis): противник измеряет мощность потребления электроэнергии каким-либо устройством (например, смарт-картой), которое производит умножение какого-либо секретного числа на точку (к примеру, генерирует ключевую пару и в процессе вычисляет открытый ключ, умножая закрытый ключ на базовую точку, – см. описание процедур генерации ключевых пар далее).

Если измерения достаточно точны, то атакующий, зная алгоритм скалярного умножения (к примеру, пусть это будет бинарный алгоритм, который мы описали ранее), может понять, в какие моменты времени карта выполняет сложение (мощность больше), а в какие – удвоение (мощность меньше), и по этой информации легко узнать все биты секретного числа.

Таким образом, очевидно, что форма Эдвардса – это весьма удобная для прикладных криптографических применений форма эллиптической кривой. Единственное ограничение – кривая Эдвардса всегда имеет порядок, который делится на 4. Это означает, что не все кривые в форме Вейерштрасса можно преобразовать в форму Эдвардса.

К примеру, это невозможно сделать для описанной выше secp256k1 – «главной» кривой систем Биткойн и Эфириум, т. к. она содержит простое число точек. Такая же ситуация с кривой bn254, ведь это тоже кривая простого порядка.

Обратное же преобразование возможно всегда: любую кривую из формы Эдвардса $ex^2 + y^2 = 1 + dx^2y^2$ можно преобразовать в форму Вейерштрасса $y^2 = x^3 + ax + b$, их коэффициенты (a, b) и (e, d) связаны следующим образом:

$$a = s^2 - 3t^2;$$

$$b = 2t^3 - ts^2,$$

где s и t зависят от e и d :

$$\begin{aligned} s &= \frac{e-d}{4}; \\ t &= \frac{e+d}{6}. \end{aligned}$$

Пусть у нас есть точка (x, y) на скрученной кривой Эдвардса $E_{e,d}^E$ с коэффициентами e и d . Как ее преобразовать в соответствующую ей точку (x', y') на кривой в форме Вейерштрасса $E_{a,b}^W$?

Подобное преобразование ($E_{e,d}^E \rightarrow E_{a,b}^W$) точки (x, y) в точку (x', y') выглядит так:

$$(x', y') = \left(s \frac{1+y}{1-y} + t, s \frac{1+y}{(1-y)x} \right).$$

Опишем обратное преобразование ($E_{a,b}^W \rightarrow E_{e,d}^E$). Пусть имеется точка (x', y') на кривой $E_{a,b}^W$, и мы хотим получить на кривой $E_{e,d}^E$ соответствующую ей точку (x, y) , тогда:

$$(x, y) = \left(\frac{x' - t}{y'}, \frac{x' - t - s}{x' - t + s} \right).$$

Кривые в форме Монтгомери

Кривая в форме Монтгомери имеет следующий вид:

$$By^2 = x^3 + Ax^2 + x,$$

где коэффициенты A и B таковы, что $B \neq 0$ и $A^2 \neq 4$ (это условие того, что данная кривая несингулярна).

Порядок группы точек кривой в форме Монтгомери всегда делится на 4, как и для кривых в форме Эдвардса. Из этого, в частности, следует, что кривые простого порядка (например, secp256k1 или P-256) невозможно представить в форме Монтгомери.

Обозначим кривую Монтгомери с коэффициентами A, B как $E_{A,B}^M$. Ее j -инвариант равен:

$$j(E_{A,B}^M) = 256 \frac{(A^2 - 3)^3}{A^2 - 4}.$$

Опишем групповой закон сложения точек для кривой в форме Монтгомери. Сумма двух точек $Q_1 = (x_1, y_1)$ и $Q_2 = (x_2, y_2)$ есть точка $Q_3 = (x_3, y_3)$, такая, что:

$$x_3 = B\lambda^2 + (x_1 + x_2) - A;$$

$$y_3 = \lambda(x_1 - x_3) - y_1,$$

где λ вычисляется различным образом при сложении и удвоении точек:

- случай $Q_1 \neq Q_2$, сложение точек:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1};$$

- случай $Q_1 = Q_2$, удвоение точки:

$$\lambda = \frac{3x_1^2 + 2Ax_1 + 1}{2By_1}.$$

Любая кривая в форме Монтгомери бирационально эквивалентна (birationally equivalent) некоторой скрученной кривой Эдвардса. Обратное также верно. Это означает, что эти кривые связаны друг с другом рациональными выражениями над их полем.

В точности то же самое нельзя сказать про связь между формой Вейерштрасса и Монтгомери (или Вейерштрасса и Эдвардса), т. к. эквивалентные кривые в форме Вейерштрасса существуют для любой кривой Монтгомери (или Эдвардса), но не для любой кривой Вейерштрасса можно найти эквивалентную кривую Монтгомери (или Эдвардса), поскольку для этого необходимо

выполнение условия, что число точек делится на 4, а кривые Вейерштрасса могут иметь порядок группы, не делящийся на 4.

Получение коэффициентов в форме Эдвардса для кривой в форме Монтгомери ($E_{A,B}^M \rightarrow E_{e,d}^E$):

$$e = \frac{A+2}{B};$$

$$d = \frac{A-2}{B}.$$

Преобразование точек в форму Эдвардса $(x, y) \rightarrow (x', y')$:

$$(x', y') = \left(\frac{x}{y}, \frac{x-1}{x+1} \right).$$

Обратное предыдущему преобразование точки из формы Эдвардса в форму Монтгомери $(x', y') \rightarrow (x, y)$:

$$(x, y) = \left(\frac{1+y'}{1-y'}, \frac{1+y'}{(1-y')x'} \right).$$

Пример кривой в форме Монтгомери и ее перевод в форму Эдвардса

Рассмотрим в качестве примера кривую Curve25519.

Это кривая в форме Монтгомери $y^2 = x^3 + 486662x^2 + x$ над простым полем с характеристикой $p = 2^{255} - 19$.

Опишем далее основные параметры данной кривой:

- коэффициенты:

$$A = 486662;$$

$$B = 1;$$

- порядок простой подгруппы:

$$q = 2^{252} + 27742317777372353535851937790883648493;$$

- cofactor = 8;

- общее число точек $\#E$ кривой Curve25519:

$$\#E = \text{cofactor} * q = 8 *$$

$$(2^{252} + 27742317777372353535851937790883648493) = \\ = 2^{255} + 221938542218978828286815502327069187944;$$

- генератор подгруппы:

$$x = 9;$$

$$y = 20AE\ 19A1\ B8A0\ 86B4\ E01E\ DD2C\ 7748\ D14C$$

$$923D\ 4D7E\ 6D7C\ 61B2\ 29E9\ C5A2\ 7ECE\ D3D9$$

(шестнадцатеричное значение).

Если Curve25519 перевести в скрученную форму Эдвардса, то мы получим:

$$-x^2 + y^2 = 1 - (121665/121666)x^2y^2.$$

Эта кривая чаще всего в литературе обозначается как Edwards25519.

2.3 Основные алгоритмы электронной подписи

Рассмотрим наиболее часто применяемые (в том числе в блокчейн-платформах) алгоритмы электронной подписи на основе эллиптических кривых.

2.3.1 Алгоритм ECDSA

Elliptic Curve Digital Signature Algorithm (ECDSA) – алгоритм электронной подписи на эллиптических кривых, который является одним из вариантов известного алгоритма Эль-Гамаля. По сути, ECDSA – это аналог широко известного алгоритма DSA (Digital Signature Algorithm), в котором мультипликативную группу конечного поля заменили на группу точек эллиптической кривой.

Алгоритм ECDSA включен во многие стандарты (ISO, IEEE, NIST и т. д.) и де-факто является наиболее часто используемым алгоритмом электронной подписи в мире криптовалют.

ECDSA включен в текущий стандарт электронной подписи США, которым является выпущенный NIST документ FIPS 186-4 Digital Signature Standard [130]. Помимо ECDSA, в этом документе есть еще и алгоритмы DSA и RSA, которые также являются стандартами подписи США.

В следующую версию стандарта, FIPS 186-5, NIST собирается добавить подпись Шнорра на скрученных кривых Эдвардса – EdDSA (Edwards-curve Digital Signature Algorithm); данный алгоритм будет рассмотрен далее.

Схема ECDSA использует параметры эллиптической кривой (коэффициенты a и b , поле кривой F_{p^n} , базовую точку P порядка q) и хеш-функцию Hash.

F_{p^n} , поле кривой для ECDSA, на практике чаще всего состоит из простого числа элементов, т. е. из всех чисел из интервала от 0 до $p - 1$, и поэтому называется простым полем (prime field) и обозначается как F_p или $GF(p)$.

В вышеупомянутом стандарте FIPS, помимо кривых над простыми полями, присутствуют также и так называемые кривые над бинарными полями (binary field), т. е. полями F_{2^n} , которые имеют характеристику, равную двум, и состоят из 2^n элементов. Но по различным причинам (в том числе из-за того, что они более медленные) они редко используются в реальной жизни.

Опишем основные операции алгоритма ECDSA.

Генерация ключевой пары ECDSA

Вход: параметры кривой.

Выход: закрытый ключ – число d , открытый ключ – точка Q .

Последовательность действий:

- 1) генерация закрытого ключа d – случайного целого числа в интервале $0 < d < q$;
- 2) вычисление открытого ключа Q – скалярное умножение закрытого ключа d на базовую точку P :

$$Q = [d]P.$$

Создание подписи ECDSA

Вход: параметры кривой и хеш-функция Hash, подписываемое сообщение m , закрытый ключ d .

Выход: подпись $(r \parallel s)$ – конкатенация чисел r и s , таких, что $0 < r < q$, $0 < s < q$.

Последовательность действий:

- 1) хеширование сообщения m и вычисление остатка от деления на q : $e = \text{Hash}(m) \bmod q$;
- 2) генерация случайного целого числа k в интервале $0 < k < q$ и вычисление точки $C = [k]P$;
- 3) вычисление $r = C_x \bmod q$, где C_x – x -координата точки C . Если $r = 0$, то возврат к шагу 2;
- 4) вычисление k^{-1} , мультипликативно обратного к k по модулю q (т. е. k^{-1} – это такое число, что $k^{-1}k \equiv 1 \pmod{q}$);
- 5) вычисление $s = k^{-1}(e + dr) \bmod q$. Если $s = 0$, то возврат к шагу 2;
- 6) подписью считается пара $(r \parallel s)$.

Проверка подписи ECDSA

Вход: параметры кривой и хеш-функция Hash, сообщение m , подпись $(r \parallel s)$ со-общения m , открытый ключ Q .

Выход: подпись верна / подпись неверна.

Последовательность действий:

- 1) проверка значений r и s подписи на принадлежность интервалу от 1 до $q - 1$. В случае если хотя бы одно из них лежит вне интервала, подпись считается неверной;
- 2) хеширование сообщения m и вычисление остатка от деления значения хеша на q : $e = \text{Hash}(m) \bmod q$;
- 3) вычисление s^{-1} , мультипликативно обратного к s по модулю q ;
- 4) умножение значения e , полученного на шаге 2, на s^{-1} : $u = es^{-1} \bmod q$;
- 5) умножение значения r из подписи на s^{-1} : $v = rs^{-1} \bmod q$;
- 6) вычисление точки $R = [u]P + [v]Q$;
- 7) вычисление $r' = R_x \bmod q$, где R_x – x -координата точки R ;
- 8) если r' равно значению r из самой подписи $(r \parallel s)$, то подпись верна, иначе – подпись неверна.

2.3.2 ГОСТ Р 34.10–2012

Алгоритм, определенный в ГОСТ Р 34.10–2012 [13], является национальным стандартом электронной подписи Российской Федерации.

В документе ГОСТ Р 34.10–2012 указывается, какие типы криптонаборов можно использовать:

- 1) порядок простой подгруппы $q: 2^{254} < q < 2^{256}$ и хеш-функция Стрибог (описана в разделе 1.3) с 256-битовым выходным значением;
- 2) порядок простой подгруппы $q: 2^{508} < q < 2^{512}$ и хеш-функция Стрибог с 512-битовым выходным значением.

Параметры кривых в самом тексте стандарта не указываются (их можно найти в рекомендациях технического комитета по стандартизации ТК-26, на-

пример в документе Р 1323565.1.024–2019 «Параметры эллиптических кривых для криптографических алгоритмов и протоколов» [43]). В ГОСТ Р 34.10–2012 есть только тестовые примеры кривых, описывающие оба случая (так называемые «короткий» и «длинный» варианты), которые позволяют проверить, правильно ли была реализована подпись в коде программы.

Основное отличие ГОСТ Р 34.10–2012 от своего предшественника ГОСТ Р 34.10–2001 [12] заключается в том, что в нем используется хеш-функция Стрибог (определенная в ГОСТ Р 34.11–2012 [15]) вместо устаревшей, определенной в ГОСТ Р 34.11–94 [14]. Кроме того, в текущем стандарте появилась возможность использовать гораздо более безопасные (но, соответственно, более медленные) криптонаборы с $q: 2^{508} < q < 2^{512}$.

Таким образом, российский стандарт ЭП предлагает два уровня безопасности:

- если нам необходим 128-битовый уровень безопасности (т. е. для атакующего необходимо $\sim 2^{128}$ операций, чтобы по открытому ключу найти закрытый), то следует использовать криптонабор с кривой, у которой порядок подгруппы $q: 2^{254} < q < 2^{256}$ в сочетании с вариантом Стрибога с 256-битовым выходным значением;
- в случае необходимости 256-битового уровня безопасности предполагается использовать кривую с $q: 2^{508} < q < 2^{512}$ и Стрибог с 512-битовым выходным значением.

Генерация ключевой пары ГОСТ Р 34.10–2012

Вход: параметры кривой.

Выход: закрытый ключ – число d , открытый ключ – точка Q .

Последовательность действий:

- 1) генерация закрытого ключа d , случайного целого числа в интервале $0 < d < q$;
- 2) вычисление открытого ключа Q : скалярное умножение закрытого ключа d на базовую точку P :

$$Q = [d]P.$$

Создание подписи ГОСТ Р 34.10–2012

Вход: параметры кривой, подписываемое сообщение m , закрытый ключ d .

Выход: подпись $(r \parallel s)$ – конкатенация чисел r и s , таких, что $0 < r < q, 0 < s < q$.

Последовательность действий:

- 1) хеширование сообщения m и вычисление остатка от деления значения хеша на q : $e = \text{Hash}(m) \bmod q$, где $\text{Hash}(m)$ – хеш-функция Стрибог. Если e равно 0, то e присваивается значение 1;
- 2) генерация случайного целого числа k в интервале $0 < k < q$ и вычисление точки $C = [k]P$;
- 3) вычисление $r = C_x \bmod q$, где C_x – x -координата точки C . Если $r = 0$, то возврат к шагу 2;
- 4) вычисление $s = (ke + dr) \bmod q$. Если $s = 0$, то возврат к шагу 2;
- 5) подписью считается пара $(r \parallel s)$.

Проверка подписи ГОСТ Р 34.10–2012

Вход: параметры кривой, сообщение m , подпись $(r \parallel s)$ сообщения m , открытый ключ Q .

Выход: подпись верна / подпись неверна.

Последовательность действий:

- 1) проверка значений r и s подписи на принадлежность интервалу от 1 до $q - 1$. В случае если хотя бы одно из них лежит вне интервала, то подпись считается неверной;
- 2) хеширование сообщения m и вычисление остатка от деления значения хеша на q : $e = \text{Hash}(m) \bmod q$, где $\text{Hash}(m)$ – хеш-функция Стрибог. Если e равно 0, то e присваивается значение 1;
- 3) вычисление $v = e^{-1} \bmod q$;
- 4) вычисление $z_1 = sv \bmod q$, $z_2 = (-rv) \bmod q$;
- 5) вычисление точки $C = [z_1]P + [z_2]Q$;
- 6) вычисление $r' = C_x \bmod q$, где C_x – x -координата точки C ;
- 7) если r' равно значению r из подписи $(r \parallel s)$, то подпись верна, иначе – подпись неверна.

2.3.3 Некоторые особенности алгоритмов ECDSA и ГОСТ Р 34.10–2012

Требования к генерации используемых случайных чисел

При создании практических реализаций ECDSA и ГОСТ Р 34.10–2012 необходимо особое внимание уделять качественной генерации случайных чисел не только для генерации ключевых пар, но и для создания подписи (т. е. числа k на шаге 2).

К примеру, для ECDSA широко известны случаи, когда в результате ошибок программистов в функции подписи иногда генерировалось одно и то же значение k . Это приводило к возможности по двум подписям разных сообщений вычислить закрытый ключ.

Действительно, пусть у нас есть две такие подписи ECDSA для разных сообщений m_1 и m_2 . Тогда они будут выглядеть так: $(r \parallel s_1)$ и $(r \parallel s_2)$, т. е. у них будут одинаковые r .

Таким образом, атакующему, для того чтобы получить приватный ключ d , надо решить систему двух уравнений в поле F_q с двумя неизвестными d и k :

$$\begin{cases} s_1 = k^{-1}(e_1 + dr) \bmod q \\ s_2 = k^{-1}(e_2 + dr) \bmod q \end{cases}$$

что достаточно элементарно.

Эта ошибка в реализации не раз «вспыхивала» на практике. Вот два самых известных случая: такую ошибку хакеры нашли в игровой приставке Sony PlayStation 3 в 2010 г., что позволило ее взломать, а в 2013 г. похожая история случилась с рядом кошельков системы Биткойн (исследователи обнаружили, что в некоторых подписях транзакций повторялись значения r).

Кроме того, необходимо отметить, что отдельные реализации ECDSA и ГОСТ Р 34.10–2012 могут оказаться уязвимыми даже для случаев, когда это на первый взгляд не совсем очевидно.

Предположим, что генератор случайных чисел – качественный, но его реализация позволяет атакующему измерить число единиц в бинарном представлении k (например, это может произойти в результате атаки по времени выполнения – timing attack). Тогда, накопив несколько сотен подписей и зная, сколько битов было равно единице в разных k для каждого случая подписи, атакующий может восстановить закрытый ключ.

Сжатие точек для кривых в форме Вейерштрасса

Для экономии пространства можно представить точку не в виде координат x, y , а в сокращенной форме: в виде x -координаты и одного бита, как показано далее.

Сжатие точки

Вход: точка (x_0, y_0) кривой $y^2 = x^3 + ax + b$ над полем F_p .

Выход: (x_0, bit) .

Последовательность действий:

- 1) если y – четное число, то $bit = 1$;
- 2) иначе, т. е. если y – нечетное число, $bit = 0$.

Значение bit необходимо для однозначного восстановления y , т. к., зная только координату x_0 , можно подставить ее в правую часть уравнения и решить его относительно неизвестного y в поле F_p :

$$y^2 = x_0^3 + ax_0 + b.$$

Поскольку решений всегда будет два: y_1 и $p - y_1$, то, чтобы определить, какое из них соответствует изначальной точке (т. е. равно y_0), нужен бит четности (y_1 и $p - y_1$ не могут быть одновременно четными или нечетными).

Восстановление точки

Вход: сжатая точка (x_0, bit) , кривая $y^2 = x^3 + ax + b$ над полем F_p .

Выход: точка (x_0, y_0) .

Последовательность действий:

- 1) решаем уравнение $y^2 = x_0^3 + ax_0 + b \pmod{p}$;
- 2) в зависимости от значения bit и четности решений y_1 и $p - y_1$ считаем изначальной точкой либо (x_0, y_1) , либо $(x_0, p - y_1)$.

Проще всего уравнение $y^2 = d$ над F_p решается, если $p = 3 \pmod{4}$. В этом случае его решения равны:

$$d^{\frac{p+1}{4}} \pmod{p}$$

и

$$p - d^{\frac{p+1}{4}} \pmod{p}.$$

Многие кривые, например secp256k1, имеют именно такой модуль $p = 3 \pmod{4}$.

Формат представления сжатых точек может различаться в различных реализациях алгоритма ЭП. Например, в системе Биткойн применяется следующий формат сжатия: бит четности кодируется как байт со значением 2 для четных y и 3 для нечетных, который конкатенируется с x (т. е. сначала идет этот байт, а потом 32 байта x).

Следует отметить, что четность – не единственный способ для кодировки y . К примеру, в бите можно закодировать «знак» y , т. е. интервал, где лежит y : $y \leq (p - 1)/2$ или $y > (p - 1)/2$. Ведь если взять любую точку (x, y) , то y -координата обратной ей точки $(x, p - y)$ будет всегда по разные стороны середины диапазона $(0, p - 1)$ (за исключением точки с $y = 0$).

2.3.4 Алгоритм EdDSA

EdDSA представляет собой детерминированную подпись Шнорра на скрученных кривых Эдвардса.

Детерминированность EdDSA заключается в том, что при подписи одинаковых сообщений на одном закрытом ключе всегда будет получаться одинаковая подпись. Это важное отличие EdDSA от недетерминированных подписей типа DSA, ECDSA, ГОСТ Р 34.10–2021 и др., где подобная ситуация означает серьезные проблемы в реализации и создает возможность утечки закрытого ключа (т. к. тогда его легко посчитать, имея лишь две подписи от разных сообщений).

Алгоритм был создан международной группой криптографов (T. Lange, D. Bernstein и др.) в 2011 г. и описан в статье [68].

EdDSA позиционируется как более безопасная и быстрая альтернатива ECDSA, т. к. он использует кривые Эдвардса и не требует генерации случайных чисел в процессе подписи, что позволяет избежать атак, аналогичных атакам на ECDSA, которые были описаны выше (т. е. EdDSA по сравнению с ECDSA имеет более высокую устойчивость к side-channel атакам и ошибкам/сбоям в генераторах случайных чисел).

Де-факто данный алгоритм электронной подписи на практике является стандартом и используется в очень большом количестве самых разных систем. Его детальное описание с тестовыми векторами можно найти в RFC 8032 [152].

Известны также различные модификации алгоритма EdDSA. Одна из них – алгоритм XEdDSA, который используется в мессенджере Signal.

В отличие от EdDSA, XEdDSA – это недетерминированный алгоритм ЭП, который основан на кривых Монтгомери. Он использует случайные числа, но безопасность в данном случае не так критично зависит от качества генератора случайных чисел, как в ECDSA, поскольку в XEdDSA случайные числа нужны скорее для защиты от атак по побочным каналам.

Генерация ключевой пары EdDSA

Вход: параметры скрученной кривой Эдвардса:

- коэффициенты кривой;
- точка B – генератор подгруппы большого простого порядка q ;
- кофактор – 2^c (т. е. общее число точек кривой равно 2^cq);
- поле кривой – F_p ,

а также хеш-функция Hash с $2b$ -битовым размером выходного значения, где $2^{b-1} > p$ (т. е. любой элемент поля F_p или точка в сжатом виде (у и бит знака x) помещаются в b бит).

Выход: закрытый ключ – число d , открытый ключ – точка A .

Последовательность действий:

- 1) генерация закрытого ключа d , случайной последовательности длиной b бит;
- 2) хеширование d : $\text{Hash}(d)$;
- 3) представление выходного значения хеш-функции в бинарном виде $(h_0, h_1, \dots, h_{2b-1})$, где h_i – i -й бит;
- 4) вычисление целого числа a из битов первой половины этой битовой последовательности (игнорируя ее первые c и последние 2 бита):

$$a = 2^{b-2} + \sum_{i=c}^{b-3} 2^i h_i;$$

- 5) вычисление открытого ключа A :

$$A = [a]B.$$

Создание подписи EdDSA

Вход: параметры кривой, хеш-функция Hash, подписываемое сообщение m , закрытый ключ d .

Выход: подпись $(R \parallel s)$ – конкатенация точки R и числа s (где $s: 0 \leq s < q$, а точка R – в сжатом виде).

Последовательность действий:

- 1) вычисление $r = \text{Hash}(h_b \parallel \dots \parallel h_{2b-1} \parallel m) \bmod q$, где $h_b \parallel \dots \parallel h_{2b-1}$ – правая половина результата хеширования d ;
- 2) вычисление $R = [r]B$;
- 3) вычисление $s = (r + \text{Hash}(R \parallel A \parallel m) * a) \bmod q$, где точки R и A – в сжатой форме;
- 4) подписью считается пара $(R \parallel s)$.

Проверка подписи EdDSA

Вход: параметры кривой и хеш-функция Hash, сообщение m , подпись $(R \parallel s)$ сообщения m , открытый ключ A .

Выход: подпись верна / подпись неверна.

Последовательность действий:

- 1) проверка, что R лежит на кривой, а s – в интервале $0 < s < q$. Иначе – подпись неверна;
- 2) вычисление $h = \text{Hash}(R \parallel A \parallel m) \bmod q$;
- 3) проверка равенства $[2^c s]B = [2^c]R + [2^c]h[A]$;
- 4) если верно это равенство, то подпись верна, иначе подпись неверна.

Некоторые особенности алгоритма EdDSA

Если в результате ошибки в реализации или сбоя в вычислениях на шаге 1 подпись злоумышленник получит две подписи для разных сообщений с одним и тем

же значением r , то он сможет посчитать закрытый ключ (как и в случае с атакой на ECDSA), решив систему двух линейных уравнений с двумя неизвестными.

Действительно, пусть $(R \parallel s_1)$ – подпись сообщения m_1 , $(R \parallel s_2)$ – подпись сообщения m_2 .

Зная A , $(R \parallel s_1)$, m_1 , $(R \parallel s_2)$, m_2 , атакующий составляет и решает систему с неизвестными a и r :

$$\begin{cases} s_1 = r + \text{Hash}(R \| A \| m_1)^* a \bmod q \\ s_2 = r + \text{Hash}(R \| A \| m_2)^* a \bmod q \end{cases}$$

– и получает закрытый ключ a .

Сжатие точки кривой в форме Эдвардса (y -координата и бит) отличается от такого для кривой в форме Вейерштрасса (x -координата и бит).

На практике EdDSA чаще всего используется с кривой Edwards25519 и хеш-функцией SHA-512. Если уровня безопасности в 128 бит кривой Edwards25519 недостаточно, то можно использовать кривую Edwards448 с уровнем безопасности ~224 бита:

$$x^2 + y^2 = 1 + (39082/39081)x^2 y^2 \bmod p,$$

где модуль кривой $p = 2^{448} - 2^{224} - 1$.

Порядок простой подгруппы для данной кривой:

$$q = 2^{446} - 8335 \text{ DC16 3BB1 24B6 5129 C96F DE93} \\ 3D8D 723A 70AA DC87 3D6D 54A7 BB0D,$$

где вычитаемое приведено в шестнадцатеричном виде.

Кофактор: 4 (т. е. $2^c = 4$).

Описание обоих кривых, как в форме Эдвардса, так и в форме Монтгомери, есть в RFC 7748 [164].

2.3.5 Алгоритм BLS

Алгоритм электронной подписи BLS (Boneh-Lynn-Shacham) появился в начале двухтысячных [87]. Данный алгоритм основан на спариваниях (см. далее) и обладает рядом весьма полезных свойств, которые отсутствуют у подписей из семейств Эль-Гамала и Шнорра, описанных ранее.

Основными параметрами алгоритма BLS являются:

- спаривание e ;
- хеш-функция Hash2Point, отображающая сообщения m в точки группы G_1 .

Спаривание (pairing) $e: G_1 \times G_2 \rightarrow G_T$ представляет собой отображение пар, состоящих из элементов групп G_1 и G_2 , в группу G_T . Коммутативные группы G_1 , G_2 , G_T имеют большой простой порядок q .

Спаривание e обладает следующими свойствами:

- 1) свойством билинейности, что означает выполнение равенств:

$$e(P + Q, R) = e(P, R) * e(Q, R);$$

$$e(P, R + Q) = e(P, R) * e(P, Q);$$

- 2) свойством невырожденности, которое означает, что для любого элемента $P \in G_1$ существует такой элемент $Q \in G_2$, что $e(P, Q) \neq 1$, и наоборот: для любого $Q \in G_2$ существует $P \in G_1$ такой, что $e(P, Q) \neq 1$.

В криптографических алгоритмах обычно используются группы G_1, G_2, G_T :

- G_1 – подгруппа порядка q кривой E_1 над полем F_p ;
- G_2 – подгруппа порядка q кривой E_2 над полем F_{p^k} , являющимся расширением поля F_p (где степень расширения k равна степени вложения (embedded degree), которая определяется тем, что q делит $p^k - 1$, т. е. в мультипликативной группе поля F_{p^k} есть подгруппа порядка q);
- G_T – мультипликативная подгруппа порядка q поля F_{p^k} .

Из первого из описанных выше свойств следует, что:

$$e([a]P, [b]Q) = e(P, [b]Q)^a = e(P, Q)^{ab} = e(P, [a]Q)^b = e([b]P, [a]Q).$$

Генерация ключевой пары BLS

Вход: параметры спаривания:

- параметры кривой $E_1(F_p)$, содержащей подгруппу G_1 простого порядка q с базовой точкой P_1 ;
- параметры кривой $E_2(F_{p^k})$, содержащей подгруппу G_2 простого порядка q с базовой точкой P_2 .

Выход: закрытый ключ – число d , открытый ключ – точка $Pub \in G_2$.

Последовательность действий:

- 1) генерация закрытого ключа d , случайного целого числа в интервале $0 \leq d < q$;
- 2) вычисление открытого ключа Pub : скалярное умножение закрытого ключа d на базовую точку $P_2 \in G_2$:

$$Pub = [d]P_2.$$

Создание подписи BLS

Вход: параметры кривой E_1 , хеш-функция Hash2Point, подписьываемое сообщение m , закрытый ключ d .

Выход: подпись σ – точка $\in G_1$.

Последовательность действий:

- 1) хеширование m : $H = \text{Hash2Point}(m), H \in G_1$;
- 2) вычисление $\sigma = [d]H$.

Проверка подписи BLS

Вход: параметры спаривания и хеш-функция Hash2Point, сообщение m , подпись σ сообщения m , открытый ключ Pub .

Выход: подпись верна / подпись неверна.

Последовательность действий:

- 1) хеширование m : $H = \text{Hash2Point}(m), H \in G_1$;
- 2) проверка равенства: если $e(H, Pub) = e(\sigma, P_2)$, то подпись верна. Иначе – подпись неверна.

Выше был приведен случай, когда открытый ключ – «большой» т. к. точка $Pub \in G_2$, а подпись σ – «маленькая» т. к. из G_1 .

Иногда может быть удобнее сделать наоборот; для этого нужно поменять G_1 и G_2 местами: при генерации пары получим точку из G_1 , а при подписи заменим хеш-функцию $\text{Hash2Point}(m)$ с выходом – точкой из G_1 на хеш-функцию с выходом – точкой из G_2 . В результате получится «маленький» открытый ключ и «большая» подпись.

Агрегация подписей BLS

Вычисление спариваний – относительно медленная (в десятки раз) по сравнению с умножением точки на число операция. Давайте познакомимся с необычными свойствами BLS, которые в некоторых случаях делают ее вне конкуренции по сравнению с более быстрыми дальными «родственниками» по эллиптической линии (ECDSA, EdDSA и т. д.).

Если есть n подписей различных сообщений на разных ключах, то это можно представить как множество $(\sigma_1, Pub_1, m_1), \dots, (\sigma_n, Pub_n, m_n)$. Все эти подписи можно объединить в одну:

$$(\sigma_1, \dots, \sigma_n) \rightarrow \sigma.$$

Эта процедура называется агрегацией подписей.

В отличие от большинства других алгоритмов ЭП, BLS позволяет выполнять агрегацию уже готовых подписей, а не только во время вычисления самой подписи. Это является значительным преимуществом данного алгоритма.

Агрегация в BLS производится с помощью сложения точек, из которых состоят отдельные подписи:

$$\sigma = \sum_{i=1}^n \sigma_i.$$

Очевидно, что эту операцию может выполнить любой пользователь при наличии уже готовых подписей.

Проверка агрегированной подписи заключается в проверке равенства:

$$e(\sigma, P) = \prod_{i=1}^n e(H_i, Pub_i),$$

где:

- $H_i = \text{Hash2Point}(m_i)$;
- Pub_i – открытый ключ, соответствующий сообщению m_i ;
- P – базовая точка подгруппы G_2 , т. е. P_2 , если подпись – «маленькая» (т. е. все $\sigma_i \in G_1$); иначе, при $\sigma_i \in G_2$, P равна базовой точке $P_1 \in G_1$.

Итак, для проверки агрегированной подписи из n разных сообщений требуется вычислить $n + 1$ спаривания. Очевидно, что это быстрее, чем проверять по отдельности все n BLS подписей, что потребует $2n$ спариваний.

В случае если все сообщения m_i одинаковые, то стоимость проверки сокращается до двух спариваний. Действительно, пусть в этом случае все хеш-коды H_i равны H , тогда благодаря свойству билинейности спариваний правая часть уравнения проверки выглядит следующим образом:

$$\prod_{i=1}^n e(H, Pub_i) = e\left(H, \sum_{i=1}^n Pub_i\right).$$

Таким образом, надо проверить равенство всего двух спариваний:

$$e(\sigma, P) = e\left(H, \sum_{i=1}^n Pub_i\right).$$

При использовании этого алгоритма агрегации необходимо помнить, что в чистом виде он уязвим к атаке типа «ключ мошенника» (rogue key), когда злоумышленник при помощи открытого ключа Pub_1 другого участника формирует используемый в атаке ключ такого вида:

$$Pub_2 = [x]P - Pub_1,$$

где x – случайное число, $0 < x < q$, а P – базовая точка, принадлежащая подгруппе, к которой принадлежат открытые ключи.

Далее злоумышленник хеширует сообщение m : $H = \text{Hash2Point}(m)$ и вычисляет $\sigma = [x]H$. После этого, предъявив σ , он может утверждать, что ничего не подозревающий владелец ключа Pub_1 тоже подписал сообщение m , ведь

$$e(\sigma, P) = e([x]H, P) = e(H, [x]P) = e(H, Pub_1 + Pub_2).$$

Для того чтобы предотвратить такую атаку, можно потребовать от владельца каждого ключа предъявлять доказательство с нулевым разглашением (Zero Knowledge Proof) того, что он знает реальный закрытый ключ, который соответствует его открытому ключу (т. е. что он обладает числом d , таким, что его открытый ключ $Pub = [d]P$).

Глава 3

Основные принципы работы блокчейн-технологий

В последнее время мы наблюдаем стремительный рост цифровых технологий. Все больше на слуху актуальность использования блокчейн-технологий и распределенного реестра. Иногда обе эти технологии отождествляют.

Однако это неправильно. Блокчейн действительно является разновидностью распределенного реестра, но не каждый распределенный реестр является блокчейном. В общем случае распределенный реестр – это база данных, распределенная между разными сетевыми узлами. Каждый узел принимает и обрабатывает информацию независимо от других узлов. Кроме того, узлы голосуют за обновления и принимают их большинством голосов в соответствии с определенным правилом (консенсусом). Как только консенсус достигнут, распределенный реестр обновляется, и последняя согласованная версия реестра сохраняется в каждом узле.

Блокчейн (от англ. *blockchain*) дословно переводится как цепочка блоков и представляет собой технологию построения распределенной базы данных, в которой все данные связаны между собой по определенному принципу. Данные в блокчейне (в отличие от распределенного реестра) не могут быть удалены. Кроме того, все данные в блокчейн-системе объединяются в блоки, и уже блоки связываются в единую цепь с помощью консенсуса.

Первым и самым известным применением технологии блокчейн стала криптовалюта биткойн (*bitcoin*). Впервые мир услышал о ней в 2008 году, когда Сатоши Накамото (Satoshi Nakamoto) опубликовал статью «Bitcoin: A Peer-to-Peer Electronic Cash System» [187], что можно перевести как «Биткойн: система цифровой пионговой наличности». Русскоязычный перевод данной статьи можно найти по ссылке [30]. Именно в работе Сатоши Накомото впервые описывались подходы к построению блокчейн-технологий.

Отметим, что до сих пор среди широкой общественности идут споры о том, что скрывается за именем Сатоши Накомото. Есть разные версии, среди которых в том числе версия о том, что группа разработчиков ядра платформы Биткойн взяла себе такой псевдоним.

Далее в этой главе мы рассмотрим основные структуры и механизмы, используемые в современных блокчейн-системах. Начнем с общего представления структур безотносительно их использования. Это связано с тем, что каждая блокчейн-система по-разному определяет наполнение: какие данные и какого объема

будут записаны в базу данных, какие механизмы консенсуса будут использованы для синхронизации узлов блокчейна, как часто происходит выработка блока, – все это зависит от предназначения блокчейн-системы и от ее архитектуры.

Блокчейн-система не обязательно реализует криптовалюту. Это могут быть любые системы, в которых требуется распределенное хранение данных. Например, можно использовать блокчейн для совершения сделок с недвижимостью. В этом случае все действия пользователей системы будут записаны в единый реестр. Никто уже не сможет удалить эти данные. Так же, как никто не сможет их подменить или, наоборот, сказать, что таких действий не совершал.

После того как мы рассмотрим основные принципы работы блокчейн-систем, мы перейдем к более подробному рассмотрению наиболее известных блокчейн-платформ, таких как Биткойн, Эфириум, Hyperledger, EOS. Блокчейн-платформы реализованы с использованием основных механизмов блокчейна, но каждая по-разному. При этом они предоставляют пользователям возможность использовать свою архитектуру либо просто для работы с блокчейн-системой, либо для разработки собственных блокчейн-решений.

3.1 БАЗОВЫЕ МЕХАНИЗМЫ БЛОКЧЕЙН-СИСТЕМ

3.1.1 Транзакции

Транзакция – это единичная запись в базе данных блокчейна. Это может быть запись о совершенном пользователем действии, запись о начислении пользователю денег (например, за то, что он создал блок, как это происходит для механизма консенсуса Proof of Work, но об этом позже), запись о каких-то изменениях параметров системы и т. д.

Транзакция состоит из нескольких полей. В каждое поле записывается информация определенного типа. Например, в качестве полей могут быть записаны имя пользователя или адрес пользователя в блокчейн-сети, идентификатор транзакции, временная метка, инструкции, по каким правилам обрабатывать транзакцию, и т. д.

Содержимое транзакций зависит от предназначения блокчейн-системы и архитектуры ее разработки. При проектировании будущей системы разработчик должен учитывать тот факт, что транзакции, попавшие в блокчейн, уже никогда не смогут быть удалены оттуда. Об этом необходимо помнить при проектировании блокчейн-систем. Если транзакции будут объемными, а система будет насчитывать много тысяч пользователей, то такая база данных в скором времени обретет очень большой размер, что будет затруднять работу с ней.

В общем виде транзакцию можно представить так, как это сделано на рис. 3.1.

Данные отправителя	[Дополнительные данные]	Содержание транзакции	Подпись
--------------------	-------------------------	-----------------------	---------

Рисунок 3.1. Общий вид транзакции

На рис. 3.1 мы можем видеть четыре основных поля. Данные отправителя – это те параметры, которые позволяют идентифицировать пользователя в блокчейн-системе.

В самом простом варианте в качестве параметра идентификации пользователя в сети может выступать открытый ключ пользователя. Однако в большинстве систем в целях экономии памяти для идентификации используется не сам открытый ключ пользователя сети, а некоторая его производная (например, хеш-код ключа).

Более того, для криптовалютных систем вообще рекомендуется каждый раз использовать новую пару ключей. В этом случае затрудняется прослеживание действий одного пользователя и, как следствие, трудно определить, каким количеством денег владеет тот или иной пользователь системы.

В зависимости от назначения транзакции она может содержать некоторые дополнительные данные, такие как метка времени, время блокировки транзакции, специальные служебные поля и флаги, а также указание на то, какими алгоритмами необходимо выполнить проверку подписи в транзакции.

Содержание транзакции также зависит от предназначения системы. Если это криптовалютная система, то данные транзакции, скорее всего, будут содержать так называемые входы (денежные источники для совершения транзакции) и выходы (между которыми распределяются денежные источники), как показано на рис. 3.2.

Идентификатор транзакции	[Дополнительные данные]	Входы	Выходы	Подпись
--------------------------	-------------------------	-------	--------	---------

Рисунок 3.2. Пример структуры криптовалютной транзакции

Транзакция, представленная на рис. 3.2, может иметь модификации. Так, например, в некоторых криптовалютных системах пользователь, сформировавший новый блок для цепочки, получает вознаграждение. В этом случае будет сформирована транзакция, которая содержит только выходы (денежные поступления данному пользователю) и не содержит входов (денежных источников). Пример такой транзакции представлен на рис. 3.3.

Идентификатор транзакции	[Дополнительные данные]	Выходы	Подпись
--------------------------	-------------------------	--------	---------

Рисунок 3.3. Криптовалютная транзакция, содержащая только выходы

Если блокчейн-система представляет собой какую-то систему учета, например систему голосования, то транзакция будет содержать данные о выполненном пользователем действии. В любом случае для подтверждения подлинности действий, совершенных автором транзакции, к наиболее значимым данным (а в идеале для всех данных транзакции) применяется электронная подпись.

Механизм электронной подписи в блокчейне обычно работает следующим образом. Сначала данные транзакции хешируются с использованием одного из алгоритмов хеширования (алгоритмы хеширования описаны в первой главе настоящей книги), а затем подписываются с использованием одного из алгоритмов электронной подписи, описанных во второй главе. Алгоритмы ЭП и хеширования, которые используются для формирования подписи, зависят от архитектуры системы и выбираются ее разработчиком.

Как правило, в блокчейн-системе у каждого пользователя есть пара ключей, соответствующих используемому алгоритму электронной подписи. Открытый ключ (или публичный ключ, от англ. Public key) пользователя является идентификатором пользователя блокчейн-системы или применяется для формирования адреса пользователя. В любом случае пользователь должен сообщить свой открытый ключ всем пользователям, с которыми он планирует взаимодействовать.

Закрытый ключ (или приватный ключ, от англ. Private key) генерируется пользователем при регистрации в системе. Ответственность за хранение закрытого ключа ложится полностью на плечи пользователя. Если закрытый ключ будет утерян, пользователь не сможет совершать действия в блокчейн-системе, так как утратит способность вырабатывать электронную подпись, соответствующую его открытому ключу.

Именно так, потеряв носитель с закрытым ключом или вследствие поломки носителя, многие пользователи теряют доступ к своим биткойнам (которые в декабре 2020 года показали рекордный рост стоимости, а в первой половине 2021 г. данный рекорд был значительно превзойден).

Подпись транзакции записывается в специально отведенное для этого поле. Существуют блокчейн-системы, в которых подписывается не вся транзакция, а только та информация, которая имеет критически важное значение. Делается это для того, чтобы не перегружать блокчейн-систему лишней работой.

Также важно отметить, что в настоящей главе мы рассматриваем базовые принципы построения отдельных элементов блокчейн-системы. В реальной жизни эти механизмы могут быть значительно сложнее. Например, в системе Биткойн сохраняется не только электронная подпись, но и набор инструкций для ее проверки.

На рис. 3.4 представлена общая схема формирования и проверки подписи транзакции.

В качестве примера, проиллюстрированного на рис. 3.4, мы рассмотрим некоторую транзакцию. Будем считать, что у создателя транзакции есть пара ключей: открытый ключ K_{pub} и закрытый ключ K_{priv} . При этом пользователь, создающий данную транзакцию, должен иметь возможность предоставить свой открытый ключ тем пользователям или узлам, которые будут выполнять проверку подписи.

Итак, для того чтобы сформировать подпись, из всех полей транзакции сначала по заданному алгоритму формируется хеш-код $H(m)$, а затем применяется алгоритм электронной подписи, где данный хеш-код подписывается с использованием закрытого ключа автора транзакции $Sig = \text{Signing}(H(m), K_{priv})$. Таким образом формируется подпись Sig , которая и помещается в самое последнее поле транзакции.

Для проверки подлинности транзакции выполняются следующие действия. Прежде всего из всех полей транзакции, за исключением поля «Подпись», вычисляется хеш-код по заданному алгоритму $H(m)$. Далее производится верификация подписи (Verification) на основе полученного хеш-кода $H(m)$, значения подписи Sig и открытого ключа отправителя K_{pub} . Алгоритм проверки напрямую зависит от используемого алгоритма электронной подписи.

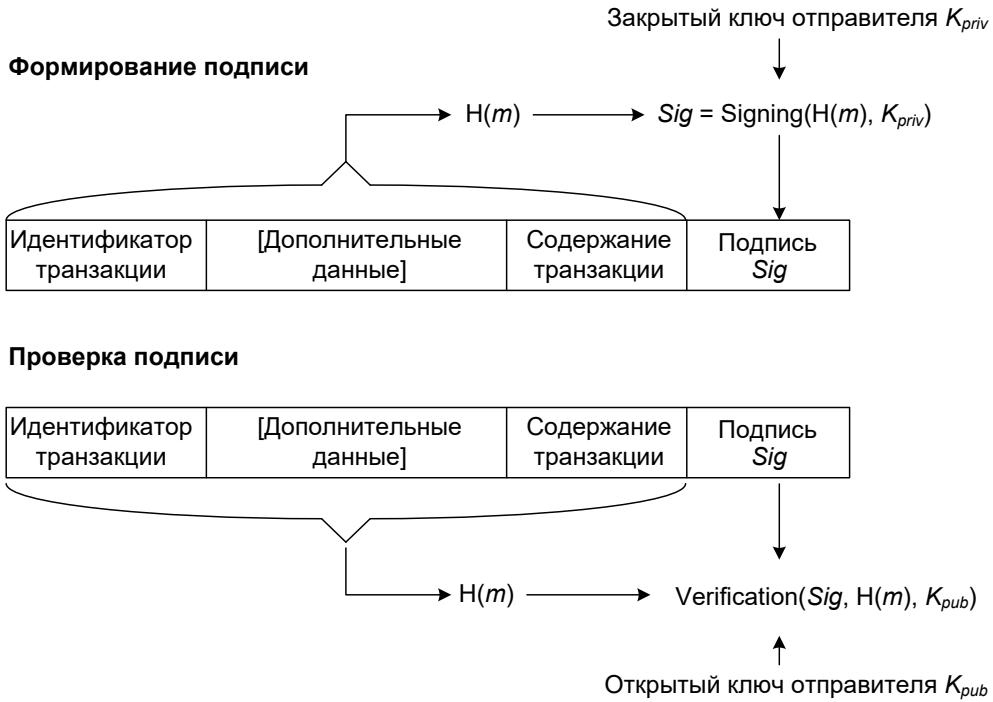


Рисунок 3.4. Формирование и проверка подписи транзакции

Если подпись транзакции оказалась верной, это значит, что транзакция прошла проверку. То есть, во-первых, все данные, содержащиеся в транзакции, не изменились с момента ее подписания, и, во-вторых, транзакция действительно сформирована пользователем, которому принадлежит закрытый ключ K_{priv} , соответствующий открытому ключу K_{pub} .

Более подробную информацию о процедурах создания и проверки электронной подписи в соответствии с различными алгоритмами можно найти во второй главе настоящей книги.

В виде транзакции может быть представлен не только денежный перевод или некоторое действие в системе, но и так называемый смарт-контракт. Смарт-контракт – это тоже транзакция, которая имеет свой идентификатор, подпись и инструкции, в соответствии с которыми программный код этого контракта будет выполняться на виртуальной машине (конечно, это применимо только для тех блокчейн-систем, которые поддерживают работу со смарт-контрактами). Смарт-контракты будут подробно описаны далее.

Все созданные и подписанные транзакции распространяются по блокчейн-сети и попадают в так называемый мемпул (от memory pool – «пул памяти»), в котором они ожидают попадания в формируемые блоки.

Необходимо упомянуть, что чем больше пользователей в блокчейн-системе, тем больше транзакций формируется пользователями и тем сложнее становится транзакциям попасть в блоки. Поэтому создатели транзакций могут предлагать некоторое вознаграждение (комиссию) тому пользователю, который поместит транзакцию в новый блок. Чем выше комиссия, назначенная за

транзакцию, тем скорее транзакция попадет в блок (на примере системы Биткойн более подробно описано в главе 4).

3.1.2 Упаковка транзакций в блоки

Мемпул представляет собой область памяти, которая разворачивается на узлах, занимающихся созданием блоков. Как было сказано выше, в мемпул попадают сформированные транзакции для их последующей упаковки в блоки.

Прежде чем поместить транзакцию в свой мемпул, узел должен выполнить ряд проверок, чтобы убедиться в валидости транзакции. В частности, узлу необходимо:

- проверить, что транзакция составлена правильно, соответствует синтаксису построения транзакций, содержит все обязательные поля и они заполнены правильными параметрами;
- если это финансовая транзакция, проверить, что списки входов и выходов транзакции существуют и не являются пустыми (бывают описанные далее исключения); также проверяется, что сумма входов и выходов совпадает;
- убедиться, что размер транзакции в байтах меньше, чем максимально допустимый размер блока (транзакции могут иметь различный размер);
- проверить, что данная транзакция ранее не была уже добавлена в пул или в ранее сгенерированный блок; отклонить транзакцию, если она уже есть в пуле или в системе;
- для финансовой транзакции проверить, чтобы ни один из ее входов не ссылался на выход другой транзакции в пуле; в противном случае отклонить транзакцию;
- если транзакция использует выход Coinbase-транзакции (такие транзакции описаны далее), необходимо убедиться, что соответствующая Coinbase-транзакция имеет минимум 100 подтверждений;
- если комиссия за транзакцию слишком мала для включения ее в блок, узел может отклонить транзакцию.

Это основной, но далеко не полный перечень проверок, которые выполняются при добавлении транзакции в мемпул. Количество проверок напрямую зависит от используемой блокчейн-системы и ее назначения.

Из мемпула транзакции собираются в блок. Размер блока зависит от того, какие настройки прописал разработчик. Например, в системе Эфириум возможно создание пустого блока, если транзакции не произошло. Кроме того, обязательно устанавливается верхний предел размера блока. Как правило, размерность блока задается в байтах.

Если в мемпуле находится небольшое количество транзакций, проблем у майнера (создателя блока) не возникает. И наоборот, чем больше транзакций в системе, тем сложнее майнерам создать блок, содержащий все транзакции.

При этом майнер не обязан добавлять в блок все ожидающие в мемпуле транзакции. В криптовалютных блокчейн-системах основным критерием попадания транзакции в блок может являться размер вознаграждения: чем большее вознаграждение майнеру прописано в виде комиссии за добавление транзакции в блок, тем быстрее эта транзакция попадет в блок.

Транзакции, не попадающие в блок, накапливаются в пуле. Проблема накопления транзакций в пуле решается так: если для общего объема транзакций, например, в 1 КБ общая сумма комиссий за все эти транзакции не превышает некоторого минимального установленного порога, то эти транзакции удаляются из пула.

Если пул заполнился полностью, то также происходит автоматическое удаление транзакций, которые имеют малую комиссию. После этого в мемпул добавляются только те транзакции, у которых размер комиссии не меньше определенного порогового значения. Это позволяет бороться с переполнениями мемпула, в том числе в результате различных вариантов атак на блокчейн-систему.

Узел, являющийся майнером и формирующий блоки цепочки, выбирает из мемпула транзакции, которые он хотел бы добавить в блок. Сколько он будет выбирать транзакций, зависит от размера блока, с которым он работает. Это значение меняется от системы к системе. Если это новая система, в ней мало пользователей, действия совершаются не очень часто, то в такой системе могут формироваться пустые блоки из одной транзакции.

Например, если посмотреть на первые блоки в системе Биткойн или Эфириум, то можно увидеть, что все первые блоки содержат только одну транзакцию – вознаграждение майнеру за добытый блок.

Система также может отслеживать среднее количество транзакций в мемпуле и на основе этого автоматически устанавливать размер блока. Кроме того, система может настраивать размер блока в зависимости от объема (веса) транзакций в мемпуле.

После того как все транзакции выбраны и проверены, из них начинает формироваться блок. Первое, что происходит с выбранными транзакциями, – из них формируется дерево Меркля (описано в следующем разделе). Дерево Меркля используется по той причине, чтобы если возникнет необходимость пересобрать блок (добавив или удалив из него транзакцию), это можно было бы сделать быстро и просто.

С помощью специальных обозревателей, например блокчейн-эксплорера (Blockchain Explorer), можно посмотреть состояние мемпула для наиболее популярных криптовалют в различное время. При этом можно посмотреть статистику как в объемах занимаемой памяти, так и в количестве транзакций.

Если сопоставить график изменения количества транзакций в мемпуле с графиком изменения стоимости биткойна, можно увидеть, что эти графики поразительным образом похожи. Чем выше стоимость криптовалюты, тем больше операций пользователи стремятся совершить, тем больше становится объем мемпула (рис. 3.5).

Данные на рис. 3.5 получены с использованием соответствующих сервисов блокчейн-эксплорера [32, 47], приведенные графики актуальны на сентябрь 2021 г.

Количество транзакций в мемпупле

Общее количество неподтвержденных транзакций в мемпупле.



Рыночная цена (USD)

Средняя рыночная цена в USD на основных биржах биткойнов.

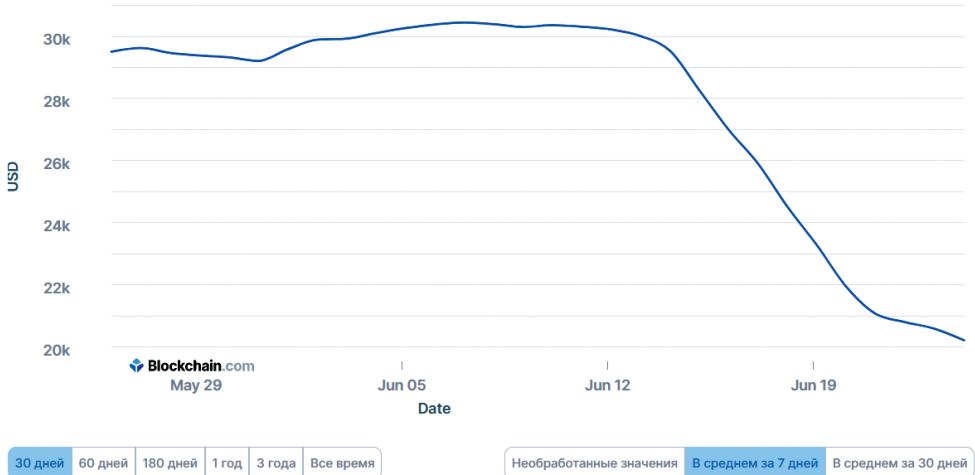


Рисунок 3.5. Количество транзакций в мемпупле (вверху) и график стоимости биткойна (внизу)

3.1.3 Применение деревьев Меркля при формировании блоков

Дерево Меркля и его использование в качестве надстройки над алгоритмами хеширования были описаны ранее в разделе 1.1.2. Дерево Меркля (или дерево хешей) широко используется в блокчейн-системах, в частности оно применяется в широко известных криптовалютных системах Биткойн и Эфириум.

Принцип построения дерева в данных системах следующий. Листьями дерева являются хеш-коды всех транзакций, которые требуется собрать в единый блок. Хеш-коды транзакций группируются попарно, и из каждой пары вырабатывается новый хеш-код. Если количество транзакций в блоке нечетное, то последняя транзакция дублируется.

Далее алгоритм повторяется, пока не останется один хеш-код, который называется корнем дерева Меркля (или корнем Меркля – Merkle root). Этот корень будет помещен в блок для проверки его целостности.

Что нам дает этот алгоритм? Допустим, что пока мы собирали блок, мы обнаружили, что в это же время другой майнер успел создать блок, в котором задействована одна из наших транзакций. Это делает наш блок невалидным. Следовательно, нам нужно удалить задействованную транзакцию из блока и пересчитать хеш-код блока. Но благодаря дереву Меркля нам не нужно пересчитывать все хеш-коды, а необходимо пересчитать только хеш-коды тех ветвей дерева, в которых использовалась удаленная транзакция.

На рис. 3.6 и 3.7 показаны примеры формирования дерева Меркля в случаях, когда возникают непарные значения. Например, если в блок помещается шесть транзакций, то на третьем уровне дерева Меркля останется хеш-значение h_9 без пары и его необходимо будет продублировать (рис. 3.6). А в случае с пятью транзакциями в правой ветке дважды останется транзакция без пары, и каждый раз ее необходимо будет дублировать (h_5 и h_8).

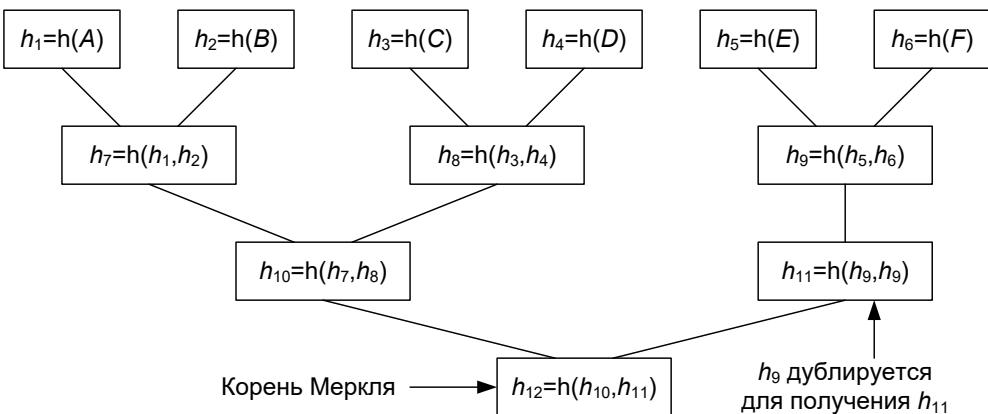


Рисунок 3.6. Дерево Меркля для шести транзакций в блоке

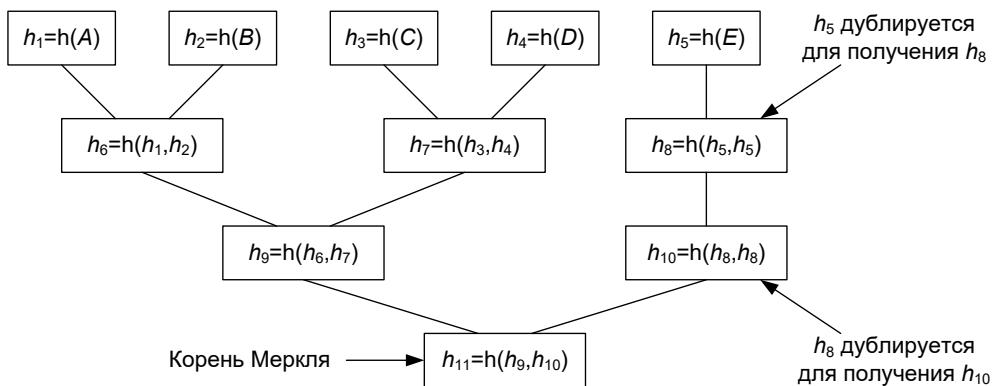


Рисунок 3.7. Дерево Меркля для пяти транзакций в блоке

Дерево Меркля позволяет клиентам системы самостоятельно проверять, была ли транзакция включена в блок, на основе значения корня Меркля из заголовка блока и списка промежуточных хеш-кодов дерева.

Например, чтобы убедиться, что для схемы на рис. 3.6 транзакция D была добавлена в блок, клиенту требуется только копия хеш-кодов h_3 , h_6 и h_{10} в дополнение к корню Меркля, все остальные вычисления клиент выполнит самостоятельно. При этом клиенту не нужно ничего знать ни о каких других транзакциях. Если бы все пять транзакций в этом блоке были максимального размера, для загрузки всего блока потребовалось бы более 500 000 байт, но для загрузки трех хеш-кодов и заголовка блока требуется всего 140 байт.

Две самые популярные на сегодняшний день платформы – Биткойн и Эфириум – используют дерево Меркля для упаковки транзакций в блок. При этом для платформы Эфириум используется сразу три дерева Меркля: одно для транзакций, второе для квитанций (квитанции содержат данные о результате выполнения транзакций, здесь речь идет о транзакциях смарт-контрактов) и третье дерево – это дерево состояний.

3.2 МЕХАНИЗМЫ КОНСЕНСУСА

Под консенсусом мы будем понимать некоторое общее правило, установленное для конкретной блокчейн-системы, в соответствии с которым происходит подтверждение записей в цепочке блокчейна для всех узлов сети.

В настоящий момент насчитывается более десятка различных консенсусов. Пожалуй, самым распространенным и самым известным является консенсус доказательства работы, который называется Proof of Work (PoW). Ниже мы рассмотрим основные принципы работы для самых известных консенсусов.

3.2.1 Консенсус доказательства работы Proof of Work

Идея, лежащая в основе механизма консенсуса с доказательством работы (от англ. Proof of Work), заключается в следующем. Цепочка блоков совместно поддерживается анонимными одноранговыми узлами в сети, поэтому требу-

ется, чтобы каждый блок имел доказательство, что в его создание был вложен значительный объем работы.

Это необходимо для того, чтобы никто не мог изменить (быстро пересчитать и переписать) исходную цепочку блоков. Объединение блоков в цепочку делает невозможным изменение транзакций, включенных в любой блок, без изменения всех последующих блоков. В результате сложность модификации конкретного блока увеличивается с каждым новым блоком, добавленным в цепочку, что усиливает эффект доказательства работы.

Вообще, понятие Proof of Work появилось задолго до появления блокчейн-систем вообще и системы Биткойн в частности. В 1992 году в работе [109] Синтия Дворк (Cynthia Dwork) и Мони Наор (Moni Naor) сформулировали следующую идею: «Чтобы получить доступ к общему ресурсу, пользователь должен вычислить некоторую функцию: достаточно сложную, но посильную; так можно защитить ресурс от злоупотребления» [51].

В качестве подобных сложных вычислительных функций могут выступать такие математические задачи, как факторизация числа (разложение произведения на два простых сомножителя), вычисление квадратного корня по модулю простого числа и некоторые другие.

Так, в качестве одной из самых распространенных задач для механизма консенсуса в блокчейн-системах (особенно это касается криптовалютных систем) выступает задача поиска хеш-кода заданной сложности. В качестве сложности обычно записывается некоторое количество нулей в начале или в конце последовательности. Так как символы хеш-кода можно представить в виде цифр шестнадцатеричного числа, то в этом случае добавление каждого нуля к заданной сложности приводит к увеличению сложности в 16 раз.

Можно провести следующий эксперимент. Давайте откроем любой онлайн-калькулятор хеш-кодов, например <https://emn178.github.io/online-tools/sha256.html>. Выберем алгоритм хеширования, например SHA-256. Введем строку для расчета и получим ее хеш-код.

Если в качестве входных данных ввести имя Евгения (ввод с клавиатуры осуществляется в текстовом формате, после ввода данные представляются в двоичном виде в соответствии с ASCII-кодом), то получается следующий хеш-код (в шестнадцатеричном виде):

5229cb8cc8c8cbb99a7b72e8efea0c195d22776ecf785029f0447f1957db01c8.

Теперь будем добавлять к имени в конце некоторое число, начиная с нуля, т. е. фактически совершать перебор значений. Наша цель – определить такое число, при котором в первом символе хеш-кода появится шестнадцатеричный 0 (табл. 3.1).

Таблица 3.1. Поиск заданного хеш-кода

Входное сообщение	Хеш-код
Евгения	5229cb8cc8c8cbb99a7b72e8efea0c195d22776ecf785029f0447f1957db01c8
Евгения0	de13ecd2329fd4411017ff070b66cc3555e292152753b150e9cf764efef1b5c7
Евгения1	471054ff614d89f8d1da9be0f78418bdb680b6a2f5bc657f004e353f66ad328a

Окончание табл. 3.1

Входное сообщение	Хеш-код
Евгения2	9193132b097df8235d07a9111467972a3eb33aef3e14fd62f5e963041eea8b8b
Евгения3	3a639225de56433daf47abe7dbd1402638c9c7e756332433ce9ef3483c31acf7
Евгения4	22e4c278433387b13a82e3648af0bdc58aa83e2feb802bb46b6199e4b154a295
Евгения5	d157086eccf9d6bb0ea5034f573feee725329aecd9dd57efc5370c1d1858c
Евгения6	769238c39df11de59fc27e30c19593a9b1cfcab0fcc7b6fe8fd699693d88262
Евгения7	9047c5aeb419178d9ebc8299e2105fa061f960110499aba4002fdb5f6e6372d9
Евгения8	2f87f217d553427c30343959e7914e0782b61b595d36615a40d9b9b1947e314e
Евгения9	9e3d1c0b899a75f7bcb887f9c83e6040218a413cf00b9e88120d5cdbfb729764
Евгения10	6cc831ace5b5b5229abeedf82f76cc08534dc1ef4ee9d538657ee159d507f782
Евгения11	b7907ba5187088149d0ba5533943022b761c98e63ae582948d6cf668646263ec
Евгения12	18d1f89f5dc276919540a88ece6bbd15668b749865b6578eed42f5f4fece9032
Евгения13	95034d35f70625f8038e3ccc4a895adab2591c4f0a020d623a4c1ae0bda0a5fb
Евгения14	bde845b11de00f37eea7d67bed4423c1967c43fa2675c7d98490ce8496825cb2
Евгения15	6c2c293ed376fd8fee31ea6170f3f31e3c9dc33bd398f59382ff77e63f806bd9
Евгения16	438622cf4acd310eaee112fb7f07f54772cd9710232a5d56876209334b04608d
Евгения17	882de9a5691be9ffe02bb5045cd2fcf6b37ec1e8c1636ac08a0e00ee1ccb9c1e
Евгения18	5b61c77dd9eedf930204e1703175f304b67c9de24effa1c88d61b241a79db1c4
Евгения19	06b604a3245191a36e112902fb85cb18250084dd97cf22ceaa2895e257fbf56d

Из табл. 3.1 видно, что при добавлении к имени значения 19 получился хеш-код, у которого первый разряд равен нулю. Если мы продолжим поиски и зададимся при этом целью найти значения, при которых хеш-код образует 2, 3, 4 нуля и т. д., мы получим результат, представленный в табл. 3.2.

Таблица 3.2. Поиск хеш-кодов разной сложности

Входное сообщение	Хеш-код
Евгения19	06b604a3245191a36e112902fb85cb18250084dd97cf22ceaa2895e257fbf56d
Евгения1391	00b99a7114904a96f023a2bec6918b529c6e01c44998e8ef6d5b3545e465a6cd
Евгения7058	0006cf8d40169151aec82c19a9020786e77207c817731e3810fe31315abe69cc

Входное сообщение	Хеш-код
Евгения 115446	0000bf801395c06cfb7a36b642fa9557ab91cf9dbdd6b5435929eaca5519f12f
Евгения 1501312	000004cd357c487d3053abab217f622772dd92b9757f61ae6798308e3fab8f5c
Евгения 14458172	000006cb31ed737fd7a37f8d456ff8742037eb329eebc378325f374694e713d
Евгения 40211489	000000005c305d46291004f0299ff5c5a638c4f76400b7d5a3ff55c7ee66ea2c

Таким нехитрым способом мы рассмотрели механизм работы консенсуса Proof of Work. Сложность работы определяется количеством нулей в искомом хеш-коде. Чем больше нулей, тем сложнее поиск. Найденное значение, при котором достигнута требуемая сложность, называется *nonce*. Процесс поиска такого хеш-кода называется майнингом, а пользователь или узел, который выполняет майнинг, называется майнером.

Ряд известных криптовалют используют в своей архитектуре механизм консенсуса Proof of Work: Биткойн, Эфириум, Litecoin, Monero, Zcash, Dogecoin и многие другие.

У процесса майнинга для консенсуса Proof of Work есть ряд недостатков. Во-первых, этот процесс требует больших вычислительных ресурсов и, как следствие, большого потребления электроэнергии. Не секрет, что в соответствии с законом Мура вычислительные ресурсы стремительно развиваются. Следовательно, должна увеличиваться сложность майнинга и потребляемая энергия.

Во-вторых, зная, какой алгоритм хеширования используют наиболее популярные криптовалютные блокчейны, производители выпускают специализированные устройства, которые справляются с решением задач майнинга гораздо быстрее, чем это может сделать процессор обычного персонального компьютера. Изначально для решения задач майнинга использовали графические процессоры (например, NVIDIA CUDA), а потом и вовсе стали выпускать интегральные схемы специального назначения – ASIC (Application-Specific Integrated Circuit).

Как было показано ранее, механизм консенсуса Proof of Work является неэффективным из-за большого количества потребляемой энергии. Кроме того, задачи майнинга усложняются с каждым годом, и поэтому майнеры уже давно не майнят поодиночке, а объединяются в так называемые пулы. При этом существует возможность того, что пулы объединятся и тогда их вычислительный ресурс позволит осуществить атаку 51 %, пересчитав всю цепочку блокчейна и внеся в нее какие-то изменения.

Атака 51 % заключается в следующем: если в руках у злоумышленника сосредоточится больше вычислительных ресурсов, чем у всей остальной сети (больше хотя бы на 1 %), тогда злоумышленник может построить новую цепочку блоков в соответствии со всеми правилами блокчейн-системы. В силу того, что злоумышленник обладает более мощным вычислительным ресурсом, рано или поздно он получит цепочку, длина которой будет больше основной цепи.

И в соответствии с правилами блокчейн-системы все пользователи системы будут вынуждены ее принять как единственную истинную.

С другой стороны, когда блокчейн-система насчитывает большое количество честных игроков, такая ситуация вряд ли случится. Честные игроки следят за размерностью пула майнеров и предпринимают меры по снижению его активности при приближении к отметке 50 % [24]. Чем больше существует блокчейн-сеть и чем больше пользователей она насчитывает, тем надежнее сохраненная в ней информация.

На базе консенсуса Proof of Work создан ряд протоколов, которые модернизируют подход Proof of Work и решают некоторые вопросы.

В качестве примера можно привести GHOST (Greedy Heaviest Observed Subtree – «самое тяжелое наблюдаемое поддерево») – алгоритм на базе Proof of Work, упрощенная версия которого используется в платформе Эфириум.

Особенность GHOST в том, что алгоритм ориентируется не на самую длинную цепочку, а на количество блоков в дереве, образуемом текущей цепочкой. Учитывается не только длина самой цепочки, но и блоки на разных ее высотах, т. е. в расчете показателей каждой цепочки фактически участвует дерево и количество блоков, которое в нем находится. Это позволяет на более раннем этапе выявить корректную цепочку блоков [91].

3.2.2 Консенсус доказательства владения долей Proof of Stake

Это, пожалуй, второй по популярности консенсус. Более того, некоторые блокчейн-системы, использующие консенсус PoW, смотрят в сторону перехода на Proof of Stake (PoS).

Впервые идея консенсуса Proof of Stake появилась в 2011 году в одном из постов известного форума bitcointalk.com. Спустя год, в 2012 году, она была реализована в криптовалюте, которая тогда называлась PPCoin, а сейчас известна под именем PeerCoin.

В консенсусе Proof of Stake нет майнеров, но есть валидаторы. Валидаторы – это пользователи блокчейн-системы, обладающие определенным ресурсом (валютой). Чем больше у пользователя ресурсов, тем больше к нему доверия и тем больше шансов, что он сможет создать новый блок.

Валидатор должен заблокировать на время часть своих ресурсов. Это своего рода ставка, которая принесет вознаграждение в том случае, если блок, созданный валидатором, будет добавлен в блокчейн-сеть. При этом вознаграждение пропорционально ставке. Так что чем выше заблокированная ставка, тем больше полученная валидатором комиссия.

С одной стороны, получается, что консенсус PoS удобнее, чем PoW, поскольку очевидна экономия вычислительного ресурса и электроэнергии. С другой стороны, участие в создании новых блоков требует наличия у валидатора больших средств. А это значит, что включиться в процесс майнинга могут только те участники, которые либо давно пользуются блокчейн-сетью и успели накопить необходимый ресурс, либо состоятельные члены, которые могут позволить себе вступить в игру с большим стартовым капиталом.

В случае с PoS для атаки 51 % пользователю придется скупить 51 % всех существующих монет. Это, в свою очередь, приведет к быстрому росту цен. Но,

с другой стороны, возникает вопрос: зачем осуществлять атаку на сеть, в которую вложено большинство средств данного пользователя?

К популярным валютам, использующим механизм консенсуса Proof of Stake, относятся такие, как Vcash, Peercoin, Stratis, BitBay, Qtum и Waves.

Для консенсуса Proof of Stake существует уязвимость, которая называется «Нечего терять» (Nothing at Stake). Пользователи, которым нечего терять, могут начать поддерживать любые блоки (в том числе и злонамеренные). Для решения этой проблемы пытаются придумать различные алгоритмы.

Известно, что платформа Эфириум давно планирует переключиться с консенсуса PoW на консенсус PoS. Однако для того, чтобы это сделать, разработчикам сначала необходимо обойти проблему «Нечего терять». Для этого предлагается использовать специальный протокол Каспер (Casper) [117]. Данный протокол отслеживает нечестных игроков системы и блокирует их, обнуляя совершенные ранее ими действия.

3.2.3 Консенсус на основе решения

задачи византийских генералов

Задача византийских генералов широко используется в различных протоколах, в том числе и криптографических. Суть задачи сводится к следующему. При ведении боевых действий различными армиями командуют различные генералы. От слаженности действий генералов зависит успех их армии.

Для координации действий генералы получают указания из центра. Если все генералы отдают единогласный приказ «В атаку!», то их армии выигрывают сражение. Если генералы отдают единогласный приказ «Отступать!», то они сохраняют свою армию и имеют время для дальнейшего маневра. Но если генералы начинают отдавать разные (несогласованные) приказы, то такая армия проигрывает сражение. Фактически она будет уничтожена по частям.

Несогласованность действий генералов может иметь две причины: либо генерал является предателем и не выполняет приказ штаба, либо кто-то намеренно изменил послание для генерала, и он получил ложный приказ. Для того чтобы генералы имели возможность согласовывать свои действия, было предложено несколько различных схем. Именно эти схемы и легли в основу консенсусов для блокчейн-сетей. Рассмотрим их более подробно.

Алгоритм консенсуса, основанный на классической задаче византийских генералов (Byzantine Fault Tolerance, BFT) [107], сводится к тому, что система продолжит устойчивую работу даже в случае, когда один или несколько узлов будут принимать решения, не совпадающие с решением большинства узлов. Различают несколько вариаций, которые описаны далее.

Практическая задача византийских генералов

Алгоритм, основанный на практической задаче византийских генералов (Practical Byzantine Fault Tolerance, PBFT) [92], – это один из первых алгоритмов, который был разработан для решения сформулированной задачи.

Идея заключается в том, что в системе существует ряд валидаторов, которые принимают решение о подтверждении транзакции (по сути, администраторы системы) [8]. Каждый валидатор должен выполнить ряд проверок и опросить

по очереди все другие узлы, чтобы принять решение, действительна транзакция или нет. Для того чтобы произошло подтверждение, необходим положительный ответ от 2/3 общего пулла участников.

Принятое решение пересыпается в сеть, с тем чтобы другие валидаторы получили к нему доступ. Консенсус достигается на основании ответов всех валидаторов. Такой алгоритм работы эффективен при низких задержках сети. Однако работа алгоритма усложняется при росте количества участников сети, так как каждое сообщение влечет за собой множество других запросов и проверок.

В настоящий момент практическая задача византийских генералов PBFT используется в следующих системах: Hyperledger, Chain, Dispatch.

Федеративное византийское соглашение

Федеративное византийское соглашение (Federated Byzantine Agreement, FBA) [138] является другим способом решения поставленной задачи.

Основа метода заключается в том, что каждый доверенный узел отвечает за свою собственную цепочку. Поэтому он проверяет все сообщения, для того чтобы установить истину. Такой подход позволяет расширять сеть, пользователи могут без проблем присоединиться к сети.

Валидаторы могут быть назначены владельцами сети или выбираться из существующих пользователей системы. В любом случае, по федеративному византийскому соглашению транзакции подтверждаются определенным количеством валидаторов, выбранных из активных в данный момент пользователей.

В настоящий момент федеративное византийское соглашение используется в таких системах, как Stellar и Ripple.

Делегированная задача византийских генералов

Делегированная задача византийских генералов (Delegated Byzantine Fault Tolerance, DBFT) [97] в полной мере реализует классическую задачу генералов. При этом система продолжает работать даже в том случае, если некоторые участники находятся онлайн.

Делегированная модель была предложена как вариант модели BFT, для которой возникают проблемы скорости верификации при увеличении числа пользователей системы. В данном случае назначаются валидаторы (те, кому делегируются права), которые проверяют транзакции, прежде чем переслать их другим узлам системы. Если делегированный участник окажется скомпрометирован, то остальные участники системы легко могут заменить его, делегировав на его роль другого участника.

Несмотря на то что протокол ориентирован на открытое использование, делегирование полномочий все же ведет к некоторой централизации.

Наиболее известным случаем применения протокола DBFT считается платформа NEO. Однако при этом в платформе NEO внесена модификация в алгоритм консенсуса таким образом, что участники системы не только выбирают (делегируют) валидаторов, но, кроме того, получают некоторый токен как часть доли от дохода выбранного ими валидатора.

3.2.4 Другие механизмы достижения консенсуса

Коротко рассмотрим прочие механизмы достижения консенсуса.

Консенсус на основе доказательства деятельности

Механизм консенсуса на основе доказательства деятельности Proof of Activity (PoA) использует элемент централизации.

Для проверки транзакций и блоков используются определенные выбранные узлы системы. Это, с одной стороны, дает большую пропускную способность системе. С другой стороны, уходит в сторону от основного постулата блокчейна – децентрализации.

Как следствие консенсус PoA можно использовать только в частных (приватных) блокчейнах. Тем не менее данный механизм реализован в тестовой сети платформы Эфириум, которая называется Kovan [19].

Консенсус на основе доказательства сжигания монет

Консенсус на основе доказательства сжигания монет Proof of Burn используется, например, в криптовалюте Slimcoin.

В основе консенсуса Proof of Burn лежит фактически та же идея, что и в основе консенсуса Proof of Work, – необходимо доказать сообществу свою приверженность сети. Только если в случае с Proof of Work пользователи делают это с помощью сжигания большого количества энергии и майнинга с использованием специального оборудования, то в случае с Proof of Burn пользователи доказывают свою приверженность, фактически растратчивая свои деньги.

Под сжиганием понимается отправка денег на несуществующий адрес сети, то есть туда, где их никто не сможет потратить. Считается, что, сжигая свои монеты, пользователи как бы вкладывают в виртуальную майнинговую мощность. Таким образом, чем больше монет сжег пользователь, тем больше виртуальных ресурсов он получил и тем быстрее он сможет создать блок.

Консенсус на основе использования направленного ациклического графа

Направленный ациклический граф (Directed Acyclic Graph, DAG) работает без учета структуры блоков самого блокчейна. Он асинхронно проверяет транзакции. По принципу DAG построены такие криптосистемы, как Iota, Hashgraph, Raiblocks/Nano.

Направленный ациклический граф состоит из направленных ребер, которые не имеют циклов. Подобные графы применяют, в частности, в технологии хеш-графов (hashgraph), которые позиционируются как улучшенный аналог блокчейн-систем [58].

Одним из самых известных механизмов консенсуса на основе направленного ациклического графа является консенсус Tangle, предложенный для Iota [200].

Для того чтобы иметь возможность сделать отправку транзакции в общую сеть, вы должны подтвердить две предыдущие транзакции. Фактически проверка два к одному должна дать уверенность в стабильности работы системы.

В то же время теоретически существует возможность проведения атаки, если какой-то узел сможет сгенерировать треть от всех транзакций. Поэтому для предотвращения подобных атак в системе Iota была предусмотрена повторная проверка специальным координатором системы. При этом разработчики системы говорят о том, что это вынужденная мера, которая будет устранена, как только сеть станет достаточно большой.

Консенсус на основе доказательства времени ставки

Еще одной из разновидностей консенсуса является консенсус на основе доказательства времени ставки (Proof of Stake Time, PoST), который определяет валидаторов не в соответствии с их текущим состоянием, а в соответствии с тем временем, в течение которого они этим состоянием владеют. То есть в данном случае чем дольше пользователь является держателем ресурсов, тем больше у него шансов создать блок.

Одним из примеров использования консенсуса на основе доказательства времени ставки является система Vericoin.

Делегированное доказательство доли владения

Делегированное доказательство доли владения (Delegated Proof of Stake, DPoS) является еще одной разновидностью консенсусов на основе владения.

Несмотря на схожие названия, DPoS сильно отличается от консенсуса PoS. В DPoS владельцы больших состояний не выполняют самостоятельно проверку блоков и транзакций. Они участвуют в выборе делегатов, которые будут выполнять проверку.

Количество делегатов может различаться в разное время и в среднем находится в диапазоне от 20 до 100. При этом делегаты периодически переизбираются. Так как делегатов сравнительно немного, то система работает достаточно быстро.

Использование консенсуса делегированного доказательства доли владения DPoS характерно для EOS, BitShares, Steemit.

Транзакция как доказательство доли

Транзакция как доказательство доли (Transaction As Proof of Stake, TAPOS) – механизм, который является по своей сути функцией программного кода платформы EOS.

Алгоритм построен таким образом, что для каждой транзакции в системе используется хеш-код последнего блока цепочки, что в свою очередь позволяет избежать повтора транзакций в сети и отследить, не находится ли транзакция в форке от основной цепочки. Все это сделано для того, чтобы противостоять возможным атакам злоумышленников.

Помимо перечисленных выше консенсусов, существуют и другие, такие как, например, консенсус на основе доказательства веса Proof of Weight (PoWeight) или консенсус на основе доказательства важности (Proof of Importance). Однако данные консенсусы пока не имеют широкого распространения и выдвигаются больше как предположения. Хотя, несомненно, находятся блокчейн-системы, которые пытаются реализовать такие консенсусы.

3.3 Выстраивание цепочки блоков

3.3.1 Принципы формирования цепочки

Основным элементом структуры блокчейн-системы является блок. Единичные действия пользователей, записанные в виде транзакций, попадают в блокчейн-систему не поодиночке, а скомпонованными группами – блоками.

То, как формируется блок, зависит от каждой конкретной блокчейн-системы. Так, например, если в сети нет активных пользователей, то могут формироваться пустые блоки, а могут не формироваться и находиться в ожидании поступления транзакций. Транзакции, которые поступают от пользователей, могут упаковываться в блоки по мере поступления (если это не финансовая блокчейн-система), а могут выбираться из общего пула транзакций в соответствии с суммой назначеннной комиссии (если это криптовалютная блокчейн-система).

Для облегчения работы с блоком для всех транзакций вычисляется корень дерева Меркля (как было описано ранее), который и помещается в блок. За счет такого подхода во время майнинга не нужно вырабатывать хеш-коды от всех транзакций, достаточно обработать все служебные поля, которые в том числе содержат корень дерева Меркля.

Максимальный размер блока может варьироваться от системы к системе. Обычно задается максимально допустимый размер в байтах. В этом случае в блок каждый раз может быть помещено разное количество транзакций, так как объем одной транзакции может значительно меняться в зависимости от количества данных (например, от количества входов и выходов в системе Биткойн).

В любом случае структура блока имеет некоторые схожие механизмы, одинаковые для всех блокчейн-систем. Блоки должны быть связаны в единую цепь, которую незаметно нарушить злоумышленнику будет очень проблематично. Для этого используется один из механизмов консенсуса, разобранный выше.

В данном разделе при рассмотрении принципов формирования цепочки блоков мы будем анализировать механизм консенсуса, основанный на доказательстве работы, то есть механизм Proof of Work. Рассмотрим цепочку блоков, представленную на рис. 3.8.

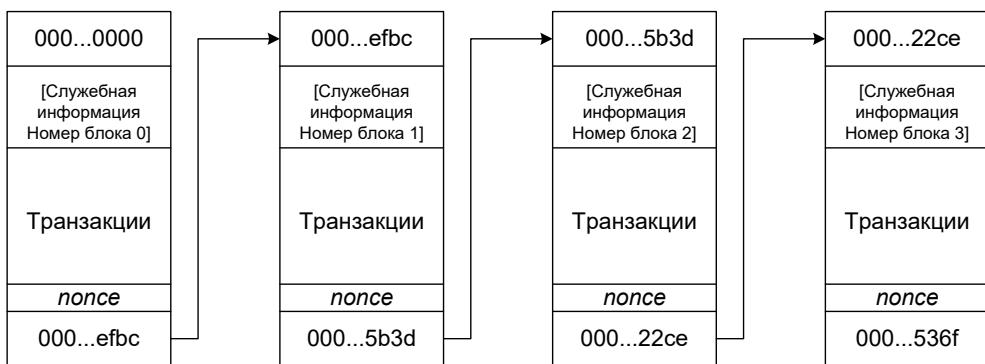


Рисунок 3.8. Пример построения цепочки блоков

Для связи блоков друг с другом в данном случае используется хеш-код, полученный от полного содержания блока. Для самого первого блока, для которого еще не существует предыдущих хеш-кодов, данное значение обычно устанавливается равным нулю.

При поиске хеш-кода вычисляющий узел (майннер) должен подобрать такое значение поля nonce, чтобы в вычисленном хеш-коде было сформировано заданное системой количество нулей.

В зависимости от блокчейн-системы принцип работы консенсуса PoW может различаться. Разработчики устанавливают среднее время, в течение которого должен быть сформирован новый блок. Так, например, в системе Биткойн каждый блок формируется раз в примерно 10 минут. При этом система периодически производит проверку времени выработки блока и в случае необходимости усложняет (увеличивает число нулей) или, наоборот, упрощает (уменьшает число нулей) решаемую задачу.

Так как со временем вычислительные ресурсы совершенствуются и наращивают производительность, то на примере биткойна можно увидеть, как изменилась сложность вычисляемого хеш-кода, притом что скорость формирования одного блока осталась неизменной – все те же 10 минут.

Проиллюстрируем различную сложность формирования блоков для разных периодов времени. На рис. 3.9 изображен одиннадцатый блок (нумерация идет от нуля) системы Биткойн [5], созданный 9 января 2009 года. Для его формирования требовался хеш-код всего с восемью нулями.

Хеш	000000002c05cc2e78923c34df87fd108b22221ac6076c18f3ade378a4d915e9
Подтверждения	742 132
Отметка времени	2009-01-09 07:05
Высота	10
Майнер	Unknown
Количество транзакций	1
Сложность	1,00
Корень Меркля	d3ad39fa52a89997ac7381c95eeffea40b66af7a57e9eba144be0a175a12b11
Версия	0x1
Биты	486 604 799
Вес	860 WU
Размер	215 bytes
Nonce (одноразовый код)	1 709 518 110
Объем транзакции	0.00000000 BTC
Вознаграждение за блок	50.00000000 BTC
Вознаграждение комиссии	0.00000000 BTC

Рисунок 3.9. Одиннадцатый блок системы Биткойн

Для сравнения на рис. 3.10 показан один из относительно недавних блоков. Блок с номером 717 000 был создан 3 января 2022 года, и его сложность составила 19 нулей [6].

Как было сказано выше, добавление одного нуля увеличивает сложность в 2^4 раза (так как хеш-коды представлены в шестнадцатеричном виде). Таким образом, увеличение сложности с 8 до 19 нулей увеличило общую сложность в 2^{44} раза!

Хеш	000000000000000000009396fbe5531a11f90bae421aa1621def645e0fe5e4e1b
Подтверждения	25 142
Отметка времени	2022-01-03 14:31
Высота	717000
Майнер	Unknown
Количество транзакций	644
Сложность	24 272 331 996 979,97
Корень Меркля	2dc9a284f44c26f4f9f9b39ef173d4e1c0ab6de15f0cd20baaca7b0b2e925605
Версия	0x20e00000
Биты	386 635 947
Вес	3 993 240 WU
Размер	1 792 542 bytes
Nonce (одноразовый код)	1 733 762 828
Объем транзакции	2240.08805058 BTC
Вознаграждение за блок	6.2500000 BTC
Вознаграждение комиссии	0.02952780 BTC

Рисунок 3.10. Блок системы Биткойн № 717 000

Итак, сложность формирования блока в блокчейн-системе устанавливается разработчиками системы и все время гибко подстраивается, с тем чтобы среднее время выработки одного блока не изменялось. Таким образом, информацию о времени выработки одного блока можно использовать для того, чтобы определить, в какое время какой из блоков появился.

По-другому дело обстоит с платформой Эфириум. Несмотря на то что Эфириум также основывается на консенсусе PoW, сложность создания блоков для майнера настолько мала, что нулей в них вы даже не заметите. Несмотря на это, сложность создания блока все-таки есть, и она заложена в базовый алгоритм. В среднем в системе Эфириум вырабатывается несколько блоков в минуту.

3.3.2 Ветвления цепочки блоков

Форки

В предыдущих разделах мы рассмотрели, как формируются транзакции, как транзакции упаковываются в блоки и как блоки выстраиваются в цепочку.

Логичным образом возникает вопрос: а что случится, если два совершенно разных майнера одновременно создадут блок, который может быть добавлен в цепочку? То есть они сформируют блок по всем правилам: соберут транзакции и определят *nonce*, который позволит получить требуемое количество нулей. Что тогда?

А тогда возникает ситуация, которая в терминологии блокчейн-систем называется форк, то есть «вилка». Фактически система разделится на две абсолютно правильные цепочки.

Те участники системы, которые получат первыми блок первого пользователя, проверят его и добавят в свою цепочку. Но тогда, получив блок второго пользователя, они забракуют его, так как хеш-код предыдущего блока не будет равен тому значению, которое использовалось для выработки блока.

Наоборот произойдет с теми участниками системы, которые первыми получат блок, сформированный вторым пользователем. Они проверят его и добавят в свою цепочку. Но тогда, получив блок первого пользователя, они забракуют его.

Таким образом возникнет два совершенно правомочных состояния системы. Система продолжит жить в расщепленном состоянии до тех пор, пока одна из цепочек не станет длиннее. В этом случае все остальные узлы после проверки должны будут принять ту цепочку, в которой оказалось большинство блоков.

Открытым остается вопрос: а что будет с теми транзакциями, которые были помещены в блоки второй цепочки, той, что в итоге оказалась в меньшинстве? Здесь есть два варианта решения вопроса.

В первом случае все транзакции будут признаны недействительными и пользователям придется формировать такие транзакции заново. Второй вариант – проверить, какие из этих транзакций смогли попасть в более длинную цепочку (ведь майнеры, формируя блоки, могут выбирать одни и те же транзакции), и вернуть в мемпул все те транзакции, для которых время жизни еще не истекло и которые на данный момент не попали в основную цепочку. На рис. 3.11 приведен пример форка.

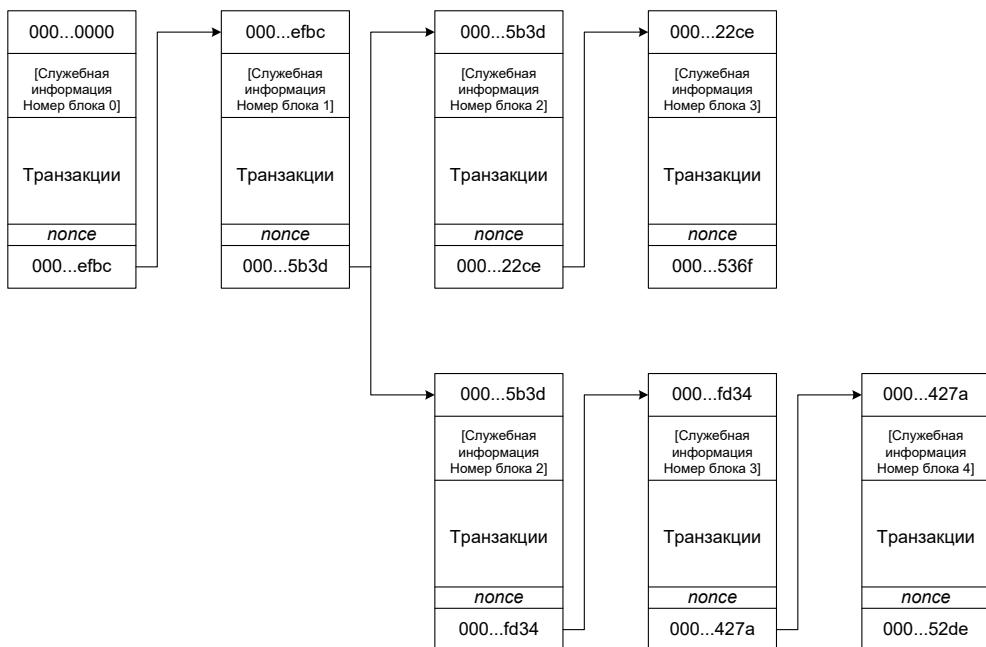


Рисунок 3.11. Пример форка для блокчейн-цепочки

Для того чтобы точно убедиться в том, что транзакция действительно попала в блок основной цепочки, необходимо выждать какое-то время. Самое

известное непредумышленное расщепление для системы Биткойн содержало в цепочке 6 блоков.

В системе Биткойн достаточно подождать, пока сформируется еще 100 блоков, для того чтобы можно было потратить монеты, полученные за создание блока. Если мы знаем, что время формирования одного блока в среднем составляет 10 минут, то можно посчитать, что временной интервал между тем, когда будут получены деньги за создание блока в системе Биткойн, и тем, когда их можно начать тратить, составит в среднем 16 часов и 40 минут.

На остальные транзакции это условие не распространяется. Но всегда необходимо помнить о возможности существования форков и отката назад. Поэтому для верности лучше выждать какое-то количество блоков, чтобы точно быть уверенными, что транзакция закрепилась в сети и ею можно распоряжаться.

В другой известной блокчейн-сети Эфириум ситуация с форками обстоит совсем по-другому. Там время создания одного блока исчисляется не минутами, а секундами, поэтому большую часть времени сеть Эфириум находится в расщепленном состоянии. Однако благодаря четко отлаженным алгоритмам работы для узлов сети проблем обычным пользователям это не доставляет.

Софтфорки и хардфорки

Известен случай, когда в результате ошибки в смарт-контракте проект The DAO лишился более 60 миллионов долларов [20], после чего сообщество пользователей сети Эфириум разделилось на два лагеря. Одни хотели вернуться к первоначальному состоянию системы. Другие хотели, чтобы совершенные ими операции за время существования проекта The DAO остались действительными.

В итоге было решено произвести форк цепочки Эфириум, в результате чего было образовано два блокчайна: Ethereum и Ethereum Classic [49]. И здесь мы подходим к другому понятию – хардфорк, или жесткое разделение.

Таким образом, форки в блокчейне могут быть также вызваны намеренным изменением программного кода, отвечающего за работу системы. При этом различают два вида форков: софтфорки и хардфорки.

В случае софтфорка программный код меняется незначительно. При софтфорке не требуется полная переконфигурация всех узлов сети, то есть можно не производить обновление программного обеспечения. При этом обычно сохраняется обратная совместимость данных.

Это означает, что новые транзакции и блоки создаются по обновленным правилам, но для проверки ранее созданных блоков учитываются старые правила. Также это означает, что новые введенные правила легко можно отменить и вернуться к старой версии работы.

Обычно софтфорк применяется в том случае, когда требуется внести какие-либо исправления или усовершенствования в работу блокчейн-сети, но при этом сохранить ее основные механизмы. Таким образом, софтфорки практически являются одним из штатных механизмов работы блокчейн-платформ.

Второй разновидностью форков является хардфорк. Хардфорк заключается в серьезном изменении кода, вследствие чего узлы со старым программным обеспечением перестают распознавать новые транзакции и блоки.

Хардфорк означает, что изменяются различные параметры формирования

цепочки, но в первую очередь изменяются правила формирования транзакций и блоков. Это означает, что программное обеспечение, которое предназначено для обработки исходной цепочки, не сможет далее поддерживать и обрабатывать цепочки, которые будут созданы в результате хардфорка.

Именно поэтому узлам блокчейн-системы так важно вовремя производить обновление программного обеспечения. Использование узлов с устаревшим программным обеспечением (необновленных узлов) может привести к распространению неверной информации и к возникновению одного из вариантов разветвления цепочки.

Это, в свою очередь, может привести пользователей к серьезным финансовым потерям. Так, например, необновленные узлы могут пересыпать и принимать транзакции, которые обновленными узлами считаются недействительными, и, следовательно, данные транзакции никогда не попадут в блоки основной цепочки.

И наоборот, необновленные узлы могут забраковать валидные блоки основной цепочки и, как следствие, отказаться их ретранслировать. Все это будет приводить к потере или искажению информации. Хардфорк применяется обычно тогда, когда разработчики хотят создать свою, новую криптовалюту на основе одной из существующих.

Обзор известных softфорков блокчейн-платформ Биткойн и Эфириум будет дан в главе 4 настоящей книги. Там же будут описаны механизмы обнаружения и противодействия нежелательным ветвлению блокчейн-цепочки.

3.4 СМАРТ-КОНТРАКТ

Считается, что идея создания смарт-контракта (или «умного контракта», от английского слова *smart* – умный) была впервые предложена Ником Сабо (Nick Szabo) в 1997 году (по некоторым источникам, идея возникла еще в 1996 или в 1994 году, но официальная статья была опубликована только в 1997 г.) [232].

Идея состояла в том, что некий программный код будет отвечать за соблюдение условий сделки разными сторонами договора. Однако на тот момент не существовало механизмов, готовых надежно обеспечить подобную идею. Все изменилось с появлением блокчейн-систем. Достоинством блокчейн-системы является тот факт, что, однажды попав в блокчейн, информация остается там навсегда и никто не может внести в нее изменения.

Широко о смарт-контрактах заговорили с появлением и широким внедрением платформы Эфириум, где специально был продуман механизм помещения подобных контрактов в блокчейн. На сегодняшний день существует целый ряд платформ, в которых возможна реализация смарт-контрактов.

Давайте рассмотрим более подробно, что же из себя представляет смарт-контракт. В силу того, что в настоящий момент отсутствуют международные правовые основы по использованию и применению блокчейн-технологий вообще и смарт-контрактов в частности, в трактовках различных государств можно встретить различные определения. Ассоциация IPChain в своем докладе по смарт-контрактам собрала более десятка определений для понятия смарт-контракт в трактовке разными странами [150].

В любом случае, все трактовки сходятся в одном мнении. Смарт-контракт должен выполняться программно и автоматически проверять заданные для него условия.

Однако многие ошибочно полагают, что смарт-контракт следит за наступлением какого-либо события и срабатывает в случае его наступления. Например, родители хотят подарить своему ребенку (скажем, сыну Роману) на совершеннолетие некоторую сумму денег, которая сейчас есть на счете одного из родителей (для определенности: на счете у папы). Общими словами мы можем сформулировать контракт так: «Дата рождения Романа 18 июля 2004 года. Я (папа) перевожу N монет на счет контракта. Когда Роману исполнится 18 лет, то перевести со счета контракта на счет Романа N монет».

Контракт подобного рода, помещенный в блокчейн-систему, не будет отслеживать текущую дату, для того чтобы перевести на счет Романа указанную сумму денег. Данное событие произойдет только в том случае, если Роман обратится к смарт-контракту с запросом «Я Роман, мне исполнилось 18 лет, переведите, пожалуйста, деньги».

В этом случае контракт проверит, во-первых, что к нему действительно обращается Роман (это можно сделать путем проверки электронной подписи), во-вторых, действительно ли с даты рождения Романа прошло 18 лет (здесь тоже есть разные способы проверки, но самым распространенным способом является сравнение времени по юникстайму (Unix time – система времени, принятая в операционных системах семейства Unix), а также сопоставление с длиной цепочки блоков, так как выработка блоков происходит систематически с определенной частотой). Если проверки будут пройдены успешно, то контракт создаст транзакцию по переводу денежных средств со счета контракта на счет Романа.

В принципе, смарт-контракт может быть применен там, где в процесс принятия решений вовлечено более одного человека. Многие запросы от бизнессообщества могут быть реализованы с применением смарт-контрактов. Рассмотрим некоторые из них в качестве примеров.

Пример первый. Система купли-продажи, причем не важно, чего. Есть продавцы, готовые что-то продать, и есть покупатели, готовые купить. Покупатель вносит свои деньги, которые получает контракт (не продавец). Продавец передает объект покупки покупателю. После подтверждения факта покупки деньги со счета контракта переходят на счет покупателя. В контракте могут быть предусмотрены различные условия. Например, скидки в зависимости от объема продажи.

Пример второй. Система доставки скоропортящихся продуктов. Есть поставщик скоропортящихся товаров. Получатель товара вносит свои деньги на оплату поставки, которые получает контракт (не поставщик). Товар можно перевозить только с соблюдением особых требований (например, влажность и температура). На всем пути транспортировки специальные датчики снимают показатели окружающей среды и записывают в блокчейн. При поступлении товара получателю цена рассчитывается автоматически с учетом показателей датчиков. Так, если условия транспортировки не были нарушены, то получателю товара будет выставлена полная стоимость. Если же условия были нарушены, то цена может снизиться либо получатель вообще сможет отказаться от получения, возможно, испорченного продукта.

Отличительной особенностью смарт-контрактов является автоматическая проверка условий и уход от посредников. Хотя в случае возникновения спорных ситуаций могут требоваться услуги арбитра – третьей независимой стороны, которая призвана решить возникший конфликт. Кроме того, контракт хранится в блокчейн-цепочке. Это значит, что он не может быть утерян и изменен, что повышает доверие к нему и, с одной стороны, повышает его надежность.

С другой стороны, контракт – это все-таки программный код. Чем больше объем кода, тем больше вероятность возникновения в этом коде ошибок. Существуют методы тестирования программных функций. Но и здесь все зависит от человеческого фактора: будут ли при тестировании предусмотрены все возможные ситуации развития событий?

Так как чаще всего контракт связан с денежными операциями, то ошибка программиста в данном случае может привести к большим потерям, связанным со многими пользователями блокчейн-сети. Наглядным подтверждением этому служит упомянутая в предыдущем разделе история, связанная с ошибкой в смарт-контракте проекта The DAO, которая привела к потере более 60 миллионов долларов.

Вообще, принципы работы и реализации смарт-контрактов различаются в зависимости от используемых блокчейн-платформ. Первые попытки реализовать смарт-контракты были предприняты для самой первой и самой известной криптовалюты – биткойна. Однако возможности смарт-контракта для системы Биткойн весьма ограничены (например, невозможно создание циклов, т. е. они не являются полными по Тьюрингу) и в большинстве своем основываются на принципах построения и обработки транзакций определенного вида, поэтому в данной системе нельзя реализовать все требуемые функции.

Для построения и обработки транзакций в системе Биткойн используется сценарный (скриптовый) язык Script, предназначенный для обработки заданной последовательности действий. Более подробно способы реализации смарт-контрактов для системы Биткойн будут рассмотрены далее в соответствующей главе.

Наибольшую популярность и распространение смарт-контракты получили с появлением блокчейн-системы Эфириум. В этой системе изначально все было продумано для того, чтобы иметь возможность создавать смарт-контракты [91].

Сам смарт-контракт является формой транзакции, которая помещается в блокчейн. При этом обязательными атрибутами такой транзакции являются два открытых (публичных) ключа. Один ключ формируется создателем контракта, а второй ключ формируется самим контрактом и фактически указывает уникальный адрес контракта в системе Эфириум.

Любой пользователь системы Эфириум может обратиться к адресу контракта и тем самым инициировать его исполнение. Для исполнения контракта используется специальная виртуальная машина Эфириум (Ethereum Virtual Machine, EVM). Именно она интерпретирует код контракта, написанный на языке Solidity, проверяет условия и формирует ответные транзакции как результат работы контракта.

В смарт-контрактах системы Эфириум нельзя добавлять новые функции, после того как контракт был активирован. Однако у программистов может оста-

ваться возможность изменять или удалять некоторые функции. Это возможно только при указании в коде контракта специальной функции SELFDESTRUCT.

Важно помнить, что контракт в системе Эфириум не может сам по себе формировать транзакции или вызывать другие контракты. Он может это делать только после того, как он был вызван (инициирован) одним из пользователей системы.

Очень понятно и наглядно о механизмах работы смарт-контрактов рассказано в статье [21].

Смарт-контракты можно создавать и для других платформ. Так, например, для платформы EOS контракты создаются на языке C++ или Rust. Во многом функционал контрактов для платформы EOS может быть соотнесен с функционалом системы Эфириум. Однако тот факт, что у этих платформ есть существенные отличия в принципах работы (консенсус, плата за транзакции и т. д.), влечет за собой различные плюсы и минусы.

Поэтому выбор платформы и языка разработки должен напрямую зависеть от желаний и возможностей заказчика. Подробный сравнительный анализ функциональных возможностей для EOS и Эфириум можно найти в статье [114].

Еще одна распространенная платформа – это приватный корпоративный блокчейн Hyperledger Fabric. Эту платформу нельзя в полной мере назвать децентрализованным блокчейном. С ее помощью можно создавать частные (приватные) сети для сообществ, в которых достаточно высокий уровень доверия.

Платформа поддерживает создание смарт-контрактов на общеизвестных языках, таких как Golang, JavaScript, Java. Смарт-контракт в терминах платформы называется чейнкодом (от англ. chaincode). В чейнкоде определяются правила формирования транзакций в зависимости от текущих условий. При этом выполнение смарт-контракта производится одновременно на нескольких независимых узлах с проверкой полученных в итоге результатов.

3.5 Основные виды блокчейн-систем

Современные блокчейн-системы делятся на публичные (открытые) и приватные (частные), которые рассмотрим далее.

3.5.1 Публичный блокчейн

Как это легко понять из названия, публичные блокчейн-системы находятся в открытом доступе, то есть представляют собой открытую сеть. Это означает, что вся информация, записанная в блокчейне, находится в открытом доступе. Любой пользователь системы может просматривать, читать и записывать данные в цепочке блоков, и эти данные доступны всем, в том числе тем, кто не является непосредственным участником системы.

Так, например, вы можете просмотреть содержимое блоков для систем Биткойн или Эфириум с помощью, в частности, упоминавшегося ранее обозревателя блокчейн-эксплорер (<https://www.blockchain.com/ru/explorer>), даже не являясь при этом непосредственным участником системы.

Публичный блокчейн является полностью децентрализованным, а значит, ни один участник системы не может контролировать формирование новых

данных в системе или изменять уже имеющиеся данные. К отличительным чертам публичного блокчейна можно отнести следующие:

- функционирование по принципу распределенного реестра: все узлы в цепочке блоков имеют право участвовать в проверке транзакций;
- открытое чтение и запись данных: любая участвующая сторона может читать, писать и просматривать данные в цепочке блоков;
- неизменность: после проверки записи ее нельзя изменить или удалить.

Публичный блокчейн целесообразно использовать в государственных секторах, таких как здравоохранение и образование. Например, медицинские учреждения могут использовать технологию блокчейн, чтобы вести учет всех своих операций. Врачи и другие специалисты могут добавлять данные о пациентах, стоимости лечения и других расходах.

При этом данные могут быть просмотрены всеми участниками блокчейна, что обеспечивает прозрачность, однако однажды добавленные данные не могут быть изменены. Также необходимо помнить о необходимости защиты персональных данных. А это значит, что, например, пациенты или пользователи в таких системах должны быть обезличены.

Одним из важных недостатков публичного блокчейна является значительная вычислительная мощность, необходимая для поддержки крупномасштабного распределенного реестра. Например, как было описано ранее, для достижения консенсуса каждый узел в сети должен решить сложную ресурсоемкую криптографическую проблему, называемую доказательством работы (консенсус PoW), чтобы гарантировать синхронизацию всех узлов.

3.5.2 Приватный блокчейн

Самым главным различием между публичным и приватным (частным) блокчейном является определение того круга лиц, которому разрешено участвовать в работе сети, формировать новые блоки (то есть фактически выполнять протокол консенсуса) и обеспечивать общую работу реестра. Если к публичному блокчейну может присоединиться любой желающий, то для приватного блокчейна требуется приглашение, которое должно быть подтверждено либо самим владельцем (инициатором) сети, либо набором правил, установленных инициатором сети.

Компании, которые используют приватный блокчейн, обычно создают сеть с контролем доступа, то есть заранее определяют, кому разрешено участвовать в сети и какие при этом транзакции может создавать данный участник сети. Механизм контроля доступа к сети может варьироваться: существующие участники могут определять будущих участников, регулирующий орган может выдавать лицензии на участие или вместо этого решения может принимать консорциум. При этом доступ к информации в определенных транзакциях будут иметь только те участники, которые указаны в данной транзакции, остальные участники не будут иметь к ней доступа.

Подобные блокчейн-системы также обеспечивают большую масштабируемость с точки зрения пропускной способности транзакций. Наглядным примером реализации приватной блокчейн-инфраструктуры является платформа Hyperledger Fabric от Linux Foundation, разработанная специально для удовлетворения корпоративных запросов.

3.6 КРИПТОВАЛЮТНЫЕ КОШЕЛЬКИ

Для того чтобы хранить приватный и публичный ключи (т. е. чтобы фактически иметь доступ к имеющимся криптовалютным средствам), существуют кошельки. Кошельки делятся на «горячие» и «холодные». Первый тип кошелька позволяет мгновенно потратить ваш ресурс. Второй обеспечивает только хранение монет и ключей.

Кошелек может быть полноценным узлом системы, т. е. программой, которая целиком выкачивает блокчейн-цепочку и с помощью которой пользователь, как полноценный узел, всесторонне взаимодействует с блокчейн-системой. А может быть легковесный кошелек, который не содержит в своем составе полной цепи блокчейна и обращается к доверенным узлам для совершения операций.

Программные кошельки могут быть как мобильными, так и десктопными приложениями. Обычно они являются легковесными и используют безопасную передачу данных для формирования какой-либо транзакции.

Кошельки бывают также аппаратного типа (например, в виде USB-устройства). Такие аппаратные кошельки обеспечивают надежное хранение секретного ключа за счет использования многофакторной аутентификации.

3.6.1 Программы-кошельки

Основными функциями криптовалютного кошелька является хранение закрытых ключей от аккаунтов пользователя и управление транзакциями для расходования средств, связанных с данными аккаунтами. При этом под расходованием мы понимаем создание и подпись транзакции.

Различные программы-кошельки могут выполнять как сразу все перечисленные функции (хранение ключей, подпись и создание транзакций), так и только одну из них. Программы-кошельки, которые предназначены для формирования транзакций (расходования средств), должны иметь возможность взаимодействовать с блокчейн-сетью, чтобы получать информацию из цепочки блоков и пересыпалть новые транзакции в сеть. Для других программ-кошельков, которые только хранят ключи или могут еще к тому же подписывать транзакции, не нужно самим взаимодействовать с сетью.

В самом простом случае кошелек представляет собой программу, которая выполняет все функции, связанные с управлением аккаунтом пользователя, а именно:

- генерирует закрытые и открытые ключи;
- выводит соответствующие открытые ключи и помогает распределять их по мере необходимости;
- отслеживает выходы, потраченные на данные открытые ключи;
- создает и подписывает транзакции, расходующие эти выходные данные;
- транслирует в сеть подписанные транзакции.

Программы-кошельки, которые работают в сложных для защиты средах, таких как веб-серверы, могут быть предназначены только для распространения открытых ключей.

Для того чтобы реализовать программу по распространению открытых ключей, можно предварительно создать базу данных с открытыми ключами или

адресами, после чего по запросу пересыпать скрипт или адрес открытого ключа, используя одну из записей базы данных.

Чтобы избежать повторного использования ключей, программа должна отслеживать уже использованные ключи. Также программа должна отслеживать, сколько осталось неиспользованных ключей, и вовремя восполнять их запас. Это возможно сделать с использованием родительского открытого ключа для создания дочерних открытых ключей.

Основным преимуществом программ-кошельков с полным спектром функций является легкость их применения. Нет необходимости иметь дополнительные средства для хранения ключей или распределения транзакций. Одна программа делает все, что нужно пользователю, чтобы получать, хранить и тратить монеты.

Однако в то же время у полнофункциональных кошельков есть и существенный недостаток, связанный с тем, что они хранят закрытые ключи на устройстве, подключенном к интернету. Компрометация таких устройств – нередкое дело, и подключение к интернету позволяет легко передать злоумышленнику закрытые ключи от взломанного устройства, а вместе с этим и права на управление всеми непотраченными выходами транзакций.

Чтобы защитить пользователя от кражи, многие программы-кошельки включают в себя функцию, позволяющую пользователям возможность шифровать файлы кошелька, содержащие закрытые ключи. Это защищает закрытые ключи, когда они не используются, но не может защитить от атаки, предназначеннной для захвата ключа шифрования или чтения расшифрованных ключей из памяти.

Для повышения безопасности закрытые ключи могут быть сгенерированы и сохранены отдельной программой-кошельком, работающей в более безопасной среде. Эти кошельки могут быть использованы только для подписи и должны работать совместно с программой-кошельком, которая взаимодействует с блокчейн-сетью.

Программы-кошельки, предназначенные только для подписи транзакций, обычно используют детерминированное создание ключей. Это означает, что из исходной (родительской) пары «открытый–закрытый ключ» в последующем создаются дочерние пары вида «открытый–закрытый ключ».

При первом запуске программа-кошелек, предназначенная только для подписи транзакций, создает родительскую пару ключей и отправляет соответствующий родительский открытый ключ в программу-кошелек,ирующую с сетью (сетевой кошелек).

Сетевой кошелек использует родительский открытый ключ для получения дочерних открытых ключей, при необходимости помогает их распространять, отслеживает выходы, потраченные на эти открытые ключи, создает неподписанные транзакции, расходящие эти выходные данные, и передает неподписанные транзакции в программу-кошелек, которая предназначена только для подписи.

После необязательного шага проверки программы-кошелька, предназначенной для подписи, использует родительский закрытый ключ для получения соответствующих дочерних закрытых ключей и подписывает транзакции, возвращая подписанные транзакции обратно в сетевой кошелек.

Затем сетевой кошелек транслирует подписанные транзакции в блокчейн-сеть [80].

3.6.2 Аппаратные кошельки

Аппаратные кошельки представляют собой устройства, предназначенные для подписания транзакций. В отличие от автономных программ, которым требуется отдельное автономное устройство, не подключенное к сети, аппаратные кошельки представляют собой специальное небольшое устройство, напрямую подключаемое к сетевому кошельку. При этом обеспечивается безопасность связи за счет встроенных защитных механизмов, а самому пользователю нет необходимости передавать данные вручную.

Рабочий процесс пользователя с использованием взаимодействующих между собой аппаратного и сетевого кошельков выглядит примерно так [80]:

1. С помощью аппаратного кошелька создаются родительские закрытый и открытый ключи. Аппаратный кошелек подключается к устройству с сетевым кошельком, чтобы последний мог получить родительский открытый ключ.
2. С помощью сетевого кошелька открытые ключи должны распространяться по сети для получения оплаты. Когда на кошелек будет получена какая-либо сумма и возникнет необходимость ее потратить, в программе сетевого кошелька вводятся данные транзакции, затем подключается аппаратный кошелек. При получении соответствующей команды от пользователя сетевой кошелек автоматически отправляет сведения о транзакции в аппаратный кошелек.
3. На экране аппаратного кошелька отображаются сведения о транзакции, которые необходимо просмотреть и проверить. Некоторые аппаратные кошельки могут запрашивать кодовую фразу или ПИН-код. Аппаратный кошелек подписывает транзакцию и загружает ее в сетевой кошелек.
4. Сетевой кошелек получает подписанную транзакцию от аппаратного кошелька и транслирует ее в сеть.

Основным преимуществом аппаратных кошельков является их высокий уровень безопасности в сравнении с полнофункциональными программными кошельками.

Пример аппаратного кошелька модели Ledger Nano X приведен на рис. 3.12.

При этом опять же главным недостатком аппаратных кошельков является неудобство их использования. Конечно, аппаратный кошелек проще в использовании, чем автономная программа-кошелек. Однако пользователю необходимо, во-первых, потратить средства на то, чтобы приобрести такой кошелек. Во-вторых, на пользователя ложится ответственность за хранение такого кошелька, потому что каждый раз, когда ему необходимо будет выполнить перевод денег, кошелек должен быть ему доступен.



Рисунок 3.12. Аппаратный кошелек Ledger Nano X

Глава 4

Основные блокчейн-платформы

Рассмотрим несколько широко известных примеров построения блокчейн-платформ на основе криптографических алгоритмов и механизмов, описанных в предыдущих главах данной книги.

4.1 Биткойн

Биткойн (англ. Bitcoin) представляет собой денежную единицу – наиболее известную и распространенную на текущий момент криптовалюту.

Это же название является синонимом для открытого программного обеспечения, которое было разработано для данной криптовалюты. Сегодня в понятие «биткойн» также вкладывается определение глобальной компьютерной сети, которая обеспечивает работу и поддержку программного обеспечения для стабильной работы с одноименной криптовалютой.

Один биткойн обозначается как 1 BTC и может быть разделен на 100 000 000 единиц [251]. Самая малая единица биткойна называется сатоши (Satoshi) в честь его создателя. Соответствие доли биткойна и количества сатоши представлено в табл. 4.1.

Таблица 4.1. Соответствие доли биткойна и количества сатоши

BTC (биткойн)	Satoshi (сатоши)	Альтернативное название и обозначение
0.00000001	1	
0.00000010	10	
0.00000100	100	1 мBTC, микробиткойн, юбит, микробит
0.00001000	1000	
0.00010000	10 000	
0.00100000	100 000	1 мВТС, миллибиткойн, мбит, миллибит
0.01000000	1 000 000	1 сВТС, центобиткойн, битцент
0.10000000	10 000 000	
1.00000000	100 000 000	

4.1.1 Введение в устройство блокчейн-системы Биткойн

Биткойн представляет собой открытую (публичную) блокчейн-сеть, в которой хранятся упорядоченные транзакции с указанием времени их создания. Каждый узел в сети Биткойн хранит ту цепочку блоков, которую он предварительно проверил.

Считается, что когда узлы хранят одинаковый набор блоков, то они находятся в консенсусе. В биткойн-сети используется консенсус доказательства работы, рассмотренный в предыдущей главе книги.

На рис. 4.1 показано упрощенное представление цепочки блоков биткойна. Транзакции блока обрабатываются с помощью дерева Меркля (деревья Меркля были описаны ранее), в результате чего вырабатывается корень Меркля, который также записывается в заголовок блока. Помимо корня Меркля, заголовок также содержит хеш-код от заголовка предыдущего блока.

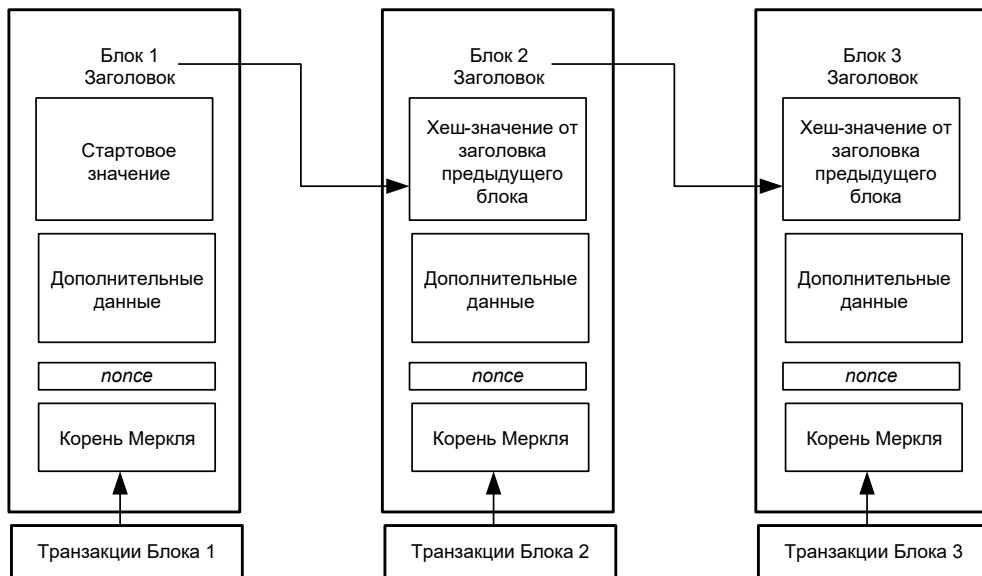


Рисунок 4.1. Упрощенное представление цепочки блоков биткойна

Все транзакции в биткойн-системе связаны между собой. На первый взгляд может показаться, что монеты просто пересыпаются из одного кошелька в другой. На самом же деле транзакция имеет строгий алгоритм построения.

У каждой транзакции есть входы и выходы. Входы – это поступление денег от ранее совершенных транзакций, выходы – это траты денег. Траты денег осуществляются на основе ранее полученных переводов от одной или нескольких транзакций. Поэтому вход одной транзакции является выходом предыдущей транзакции. При этом одна транзакция может иметь сразу несколько выходов, то есть перевод монет осуществляется сразу на несколько разных адресов. Но выход из каждой транзакции может использоваться строго один раз. Это сделано для того, чтобы избежать двойной траты (то есть одни и те же монеты нельзя потратить дважды).

У каждой транзакции есть свой идентификатор *Idtx*, который по сути представляет собой хеш-код – результат двойного хеширования транзакции.

В качестве функции хеширования дважды используется алгоритм SHA-256, подробно описанный в первой главе книги. И выходы каждой транзакции привязаны к ее идентификатору.

Получается, что все транзакции, которые записаны в биткойн-сети, могут иметь один из двух статусов: уже потраченный выход или еще не израсходованный выход (обозначается как UTXO, от англ. Unspent Transaction Output). Чтобы перевод денег в биткойн-сети был действительным, все входы транзакции должны находиться в статусе неизрасходованных (UTXO).

При проверке валидности транзакции обязательно проверяется выполнение условия, что сумма всех выходов не превышает суммы всех входов. Если это условие не выполняется, то транзакция отклоняется. Если же потраченная сумма оказывается меньше, чем составляет общая сумма входов, то есть два пути расходования остатка: остаток может быть переведен на адрес инициатора транзакции (то есть как бы осуществляется перевод сдачи) либо остаток может использоваться как комиссия тому майнеру, который замайнит данную транзакцию в блок.

На рис. 4.2 показан пример трат и поступлений для 7 транзакций. На вход самой первой транзакции с номером 0 поступил перевод в 1000 сатоши. Далее каждая транзакция тратит на 100 сатоши меньше. Таким образом, вознаграждение за каждую из этих транзакций добавит майнеру 100 сатоши.

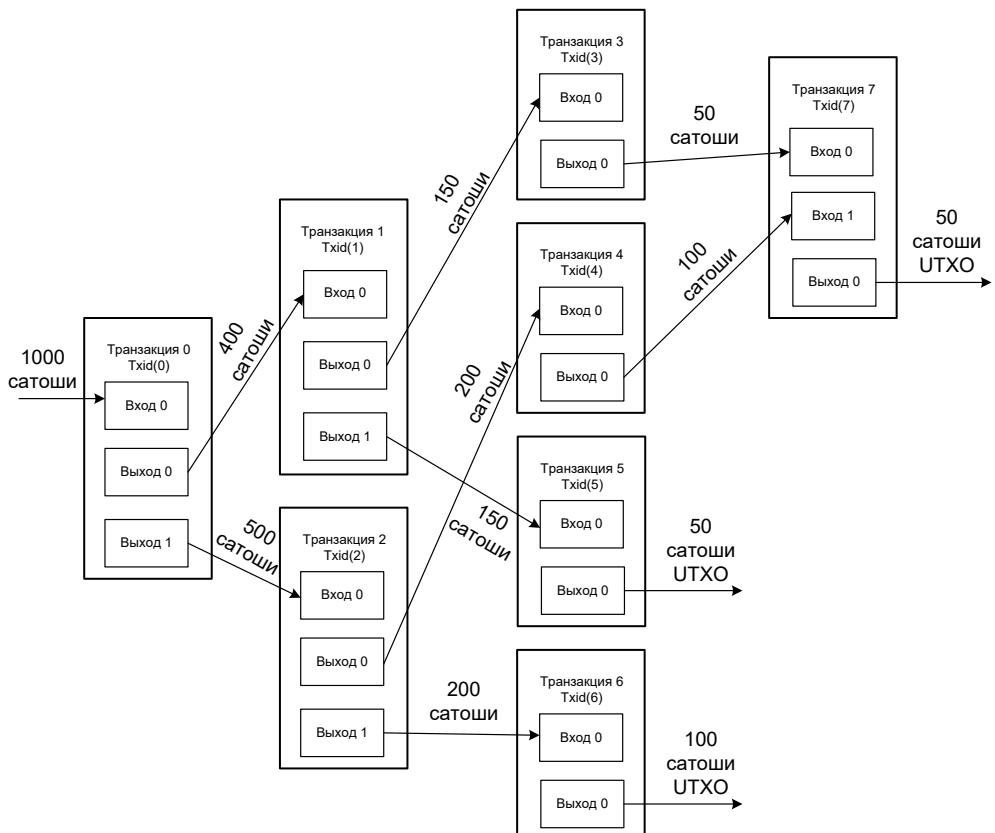


Рисунок 4.2. Пример связей между поступлениями и тратами для нескольких транзакций

По рис. 4.2 можно видеть, что для трех транзакций под номерами 5, 6 и 7 остались неизрасходованные выходы, которые, соответственно, содержат 50, 100 и 50 сатоши.

Следует отметить, что комиссия (вознаграждение) за каждую транзакцию автоматически вычитается из суммы транзакции и затем добавляется к общей сумме комиссии майнера. Так было не всегда. До определенного времени транзакции в биткойн-сети обрабатывались равновероятно и совершенно бесплатно.

4.1.2 Особенности механизма консенсуса в системе Биткойн

В биткойн-сети используется консенсус доказательства работы PoW, который ранее был рассмотрен в третьей главе. Сложность генерации нового блока зависит от количеством нулей, которые необходимо получить в вырабатываемом хеш-коде. При этом структура блока в биткойн-сети организована так, что в заголовке блока есть специальное поле nonce, которое легко получить.

Для доказательства работы вырабатывается хеш-код только для 80-байтового заголовка блока. Таким образом, тот факт, что сам блок имеет большой объем и содержит большое количество транзакций, не оказывает влияния на скорость получения хеш-кода заданной сложности.

Если же осуществляется добавление дополнительных транзакций в блок, то при этом достаточно пересчитать хеш-коды для дерева Меркля и определить его корень. Таким образом обеспечивается доказательство работы, проделанной узлом, сгенерировавшим блок.

Пересмотр сложности генерации блока происходит через каждые 2016 блоков. Для этого используются временные метки из заголовков блоков, по которым определяется, за какое время были получены последние 2016 блоков. Идеальным считается значение 1 209 600 секунд, то есть блок формируется один раз в 10 минут.

При этом если по какой-то причине для генерации последних 2016 блоков потребовалось менее двух недель, то сложность генерации хеш-кода будет автоматически скорректирована путем такого увеличения, которое (в теории) позволяет сгенерировать следующие 2016 блоков ровно за две недели. Если же генерация 2016 блоков займет больше двух недель, то сложность автоматически понизится аналогичным образом.

Согласно данным с сайта [79], в реализации ядра биткойн-сети была допущена ошибка, по которой временные метки учитываются только для последних 2015 блоков, что вносит небольшие искажения при расчете среднего времени выработки одного блока.

4.1.3 Форки в системе Биткойн

Сколько блоков может быть в цепочке и как образуется форк

Все блоки в цепи биткойна имеют свой номер, который для структуры биткойн-цепочки называется «Высота». Стартовый генезис-блок имеет высоту

ту, равную 0. Это можно проверить, перейдя на сайт по адресу https://www.blockchain.com/explorer?view=btc_blocks и указав номер блока 0 (рис. 4.3). Дальше, последовательно переключая блоки, можно видеть, как с каждым блоком высота увеличивается на 1.

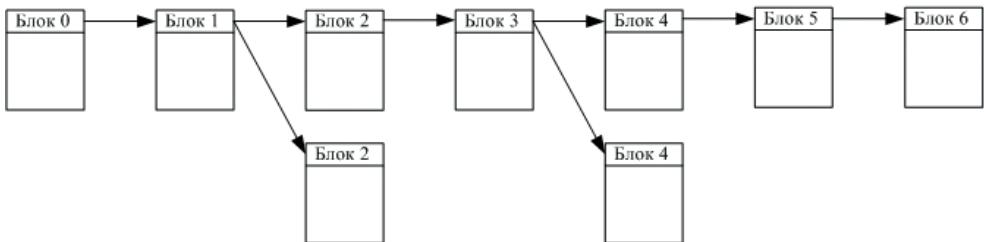
Хеш	000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f	
Подтверждения	742 142	
Отметка времени	2009-01-03 21:15	
Высота	0	
Майнер	Unknown	
Количество транзакций	1	
Сложность	1,00	
Корень Меркля	4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b	
Версия	0x1	
Биты	486 604 799	
Вес	1140 WU	
Размер	285 bytes	
Nonce (одноразовый код)	2 083 236 893	
Объем транзакции	0.00000000 BTC	
Вознаграждение за блок	50.00000000 BTC	
Вознаграждение комиссии	0.00000000 BTC	

Рисунок 4.3. Первый блок биткойн-цепочки

Любой майнер, который смог создать блок в соответствии с заданным консенсусом (то есть в случае с биткойн-цепочкой смог подобрать значение *nonce*, которое приводит к выработке хеш-кода в соответствии с заданной сложностью), может поместить этот блок в цепочку, а также сообщить об этом остальным пользователям блокчейн-системы.

При этом может получиться так, что разные майнеры одновременно сформируют разные блоки с указанием одной и той же высоты. В этом случае блокчейн-цепочка окажется в расщепленном состоянии (рис. 4.4). Фактически это происходит постоянно, то есть постоянно случаются ситуации, когда происходит раздвоение цепи. Это и является форком.

Обычные ситуации с форками



Редкие случаи больших форков

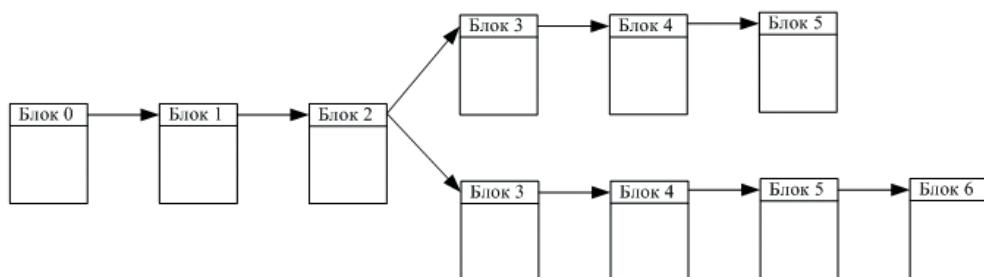


Рисунок 4.4. Примеры форков

Как же биткойн-цепь справляется с данной проблемой? Рассмотрим этот вопрос подробнее.

Майнер, сформировав новый блок, отправляет информацию о блоке в общую сеть. Каждый узел сам решает, какой из полученных блоков ему принять. Обычно обрабатывается тот блок, который узел получил первым. Таким образом на разных узлах могут сформироваться разные цепочки блоков. Та цепочка, которая оказывается длиннее других, и будет считаться основной.

Обычно расщепление бывает на уровне 1–2 блоков и эта ситуация легко разрешается (верхнее изображение на рис. 4.4).

Появление нетипичной ситуации с форком большой длины может возникнуть тогда, когда часть майнеров пытаются атаковать сеть, применив, например, атаку 51 %. Пример долгосрочного форка приведен на нижнем изображении рис. 4.4.

Так как ситуация с расщеплением цепочки является обычной для блокчейн-сети, то очень часто сформированные блоки на разных узлах могут иметь одинаковое значение высоты, но при этом разное содержание. Поэтому не рекомендуется использовать высоту блока в качестве глобального уникального идентификатора. Вместо этого на блоки обычно ссылаются по хеш-кодам их заголовков (часто с обратным порядком байтов и в шестнадцатеричном формате).

Обнаружение форков

Необновленные узлы могут забраковать валидные блоки основной цепочки и, как следствие, отказаться их ретранслировать. Все это будет приводить к по-

тере или искажению информации. Хардфорк применяется обычно тогда, когда разработчики хотят создать свою, новую криптовалюту.

Ядро платформы Биткойн (Bitcoin Core) содержит модуль, предназначенный для обнаружения хардфорков. Делается это на основе анализа заголовков в цепочке блоков. Узлы постоянно взаимодействуют друг с другом, проверяя правильность построения цепочки в соответствии с консенсусом, а также определяя, у кого цепочка блоков длиннее. Если при анализе выясняется, что у соседнего узла намного больше заголовков для цепочки блоков (по крайней мере, больше, чем на 6), то такой узел посыпает предупреждение, что необновленный узел не может переключиться на наиболее подходящую цепочку блоков. Если установлено соответствующее программное обеспечение, то узел может автоматически произвести обновление.

Изменения, вносимые в правила работы блокчейн-системы, могут быть активированы различными способами. В течение первых двух лет существования системы Биткойн было сделано несколько softфорков путем небольших изменений в клиенте с сохранением обратной совместимости. При этом вносимые изменения начинали действовать сразу после обновления.

Несколько softфорков, таких как, например, BIP30 (о BIP будет рассказано далее), начали свою работу не сразу, а по достижении определенного дня. В этом случае новое правило работы блокчейн-системы начинает применяться либо в заранее установленное время, либо при достижении определенной высоты блока.

Такие форки, начинающие свою работу при достижении определенного условия, известны как активируемые пользователем softфорки (UASF, User Activated Soft Forks), поскольку они зависят от наличия достаточного количества пользователей (узлов) для обеспечения соблюдения новых правил.

Более поздние softфорки начинали вступать в действие тогда, когда большая часть (от 75 % до 100 %) суммарной вычислительной мощности майнингового оборудования (хешрейта) сигнализирует о своей готовности к применению новых правил. Как только сигнальный порог будет пройден, все узлы начнут применять новые правила. Такие форки известны как softфорки, активируемые майнером (MASF, Miner Activated Soft Forks), поскольку их активация зависит от майнеров.

Предложения по улучшению системы Биткойн (Bitcoin Improvement Proposal, BIP) BIP16, BIP30 и BIP34 были реализованы как изменения, которые могли привести к softфоркам. Улучшение BIP50 описывается как случайный хардфорк, разрешенный путем временного понижения возможностей обновленных узлов, и как преднамеренный хардфорк, когда временное понижение версии было удалено. В документе от Гэвина Андресена (Gavin Andresen) [56] описывается, как могут быть реализованы будущие изменения правил.

4.1.4 Транзакции

Запись транзакций в блоки

Для системы Биткойн нельзя создать пустые блоки, то есть блоки, не содержащие транзакций. Каждый блок должен содержать как минимум одну транзакцию. Если других действий в системе не производилось, то в блоке будет

содержаться единственная транзакция, которая начисляет вознаграждение создавшему ее майнеру. При запуске системы данное вознаграждение составляло 50 биткойнов.

С использованием любого сервиса по просмотру блоков биткойн-цепочки, например расположенного по адресу https://www.blockchain.com/explorer?view=btc_blocks, можно увидеть, что первые созданные блоки преимущественно содержали только эту единственную транзакцию. Такая транзакция называется транзакцией генерации блока. Помимо вознаграждения за созданный блок, она также может содержать комиссии, которые пользователиплачивают за включение других транзакций в блок.

Так, например, в блоке с высотой 100 содержится всего одна транзакция, она же транзакция генерации блока. Других транзакций в блоке нет, и потому майнер не получает комиссию (см. рис. 4.5).

Вознаграждение за блок	50.00000000 BTC
Вознаграждение комиссии	0.00000000 BTC

Транзакции блока ⓘ

Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 134 bytes)	50.00000000 BTC
Хеш	2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc9020...	2009-01-11 11:00
COINBASE (Недавно Сгенерированные Монеты)	→	13A1W4jLPP75pzvn2qJ5kyyqG3qPSpb9jM 50.00000000 BTC

Рисунок 4.5. Транзакции в блоке биткойна с высотой 100

А в блоке с высотой 200 000, помимо транзакции генерации блока, содержится еще 387 дополнительных транзакций. Поэтому майнер получает комиссию за транзакции, помещенные в блок (рис. 4.6).

Вознаграждение за блок	50.00000000 BTC
Вознаграждение комиссии	0.63517500 BTC

Транзакции блока ⓘ

Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 192 bytes)	50.63517500 BTC
Хеш	dbaf14e1c476e76ea05a8b71921a46d6b06f0a950f17c5f9f1a03b8fae467...	2012-09-22 14:45
COINBASE (Недавно Сгенерированные Монеты)	→	1MdYC22GmjP2ejlVPCxyYjFyWbQCYTGHgQ8 50.63517500 BTC
Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 226 bytes)	102.12000000 BTC
Хеш	ee475443f1fbffff84ffba43ba092a70d291df233bd1428f3d09f7bd1a6054a1f	2012-09-22 14:45
1CfD77hupeUvFwBPxZ2fA8iyWmVwQY22oh	102.12000000 BTC	1FhNPPrh1TxVidoKkWFpdM5KRXw9vG1KUb 55.86000000 BTC
	→	1QE7Tazy1uHD1utCb1fCW6alF3RZ1sNwd2 46.26000000 BTC

Рисунок 4.6. Транзакции в блоке биткойна с высотой 200 000

Как было сказано ранее, при возникновении непотраченного остатка UTXO алгоритмы биткойн-системы работают таким образом, что не дают его потратить до тех пор, пока после возникновения этого остатка не будет получено 100 блоков.

Это сделано для того, чтобы избежать так называемой двойной траты, а также для того, чтобы убедиться, что неизрасходованная транзакция попала в основную цепочку блоков (а не в цепочку-форк).

Технически майнеру не обязательно искать такой блок, в котором были бы дополнительные транзакции. Однако в погоне майнера за дополнительным вознаграждением в виде комиссии за транзакции, помещенные в блок, количество транзакций в блоке может достигать нескольких тысяч. В этом случае суммарная комиссия может содержать десятые доли биткойна.

На рис. 4.7 приведен пример блока биткойна с высотой 599 999. Здесь вознаграждение за полученный блок составляет 12,5 BTC, а комиссия за 3394 транзакции – 0,248659028 BTC.

Хеш	000000000000000000003ecd827f336c6971f6f77a0b9fba362398dd867975645
Подтверждения	118 342
Отметка времени	2019-10-19 03:02
Высота	599999
Майнер	Poolin
Количество транзакций	3 394
Сложность	13 008 091 666 971,90
Корень Меркла	ca13ce7f21619f73fb5a062696ec06a4427c6ad9e523e7bc1cf5287c137ddcea
Версия	0x2000e000
Биты	387 294 044
Вес	3 993 365 WU
Размер	1 326 983 bytes
Nonce (одноразовый код)	687 352 075
Объем транзакции	26944.05623014 BTC
Вознаграждение за блок	12.50000000 BTC
Вознаграждение комиссии	0.24865028 BTC

Рисунок 4.7. Данные блока биткойна с высотой 599 999

Все транзакции в биткойн-системе, включая транзакции с вознаграждением за вновь созданный блок, представляются в формате необработанной дво-

ичной транзакции. Такое представление транзакции хешируется для создания идентификатора транзакции (обозначается как Idtx или txid).

Из всех идентификаторов транзакций строится дерево Меркля путем объединения каждого txid с другим txid и последующего хеширования их вместе. Если число идентификаторов нечетное, то последний идентификатор дублируется, то есть хешируется с копией самого себя.

Есть одно важное «но»! В 2012 году было выяснено, что реализация, которую система Биткойн использует для вычисления корня Меркля в заголовке блока, ошибочна в том, что можно легко создать несколько списков хеш-кодов, для которых будет получен один и тот же корень Меркля.

Если в блок поместить транзакции с идентичными идентификаторами txid , то существует вероятность того, что дерево Меркля будет иметь коллизию с похожим блоком, в котором такие транзакции встречаются в одном экземпляре.

Например, вычисления корня Меркля для трех транзакций ($[A, B, C]$) и корня Меркля для четырех транзакций ($[A, B, C, C]$) будут давать одинаковый результат (рис. 4.8). Это потому, что на каждой итерации хеш-функция дополняет свой промежуточный список хеш-кодов последним хеш-кодом, если список имеет нечетную длину, чтобы сделать его четной длины.

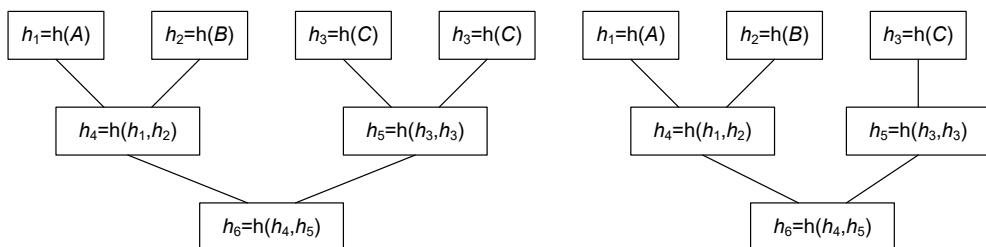


Рисунок 4.8. Пример коллизии для дерева Меркля

Получается, что для любого блока с нечетным количеством транзакций очень просто подобрать коллизию: необходимо удвоить последний элемент [81].

В случае использования честного программного обеспечения и честного взаимодействия со стороны всех участников биткойн-системы наличие возможных коллизий не влечет за собой сбоев в работе. Однако в случае нечестной игры или применения модифицированного программного обеспечения может получиться так, что в блоки попадут транзакции с расходованием одних и тех же монет (двойное расходование), при этом правильно построенные блоки без двойного расходования средств будут признаваться недействительными.

Это можно использовать для форка блокчайна, в том числе для глубоких атак с двойным расходованием средств. Именно такая уязвимость была замечена в 2012 году и внесена в общий список под названием CVE-2012-2459 [98]. Проблема была исправлена Гэвином Андресеном путем отклонения блоков с повторяющимися транзакциями и предотвращения их хеширования вообще.

Общая структура транзакций

В криптовалютных системах, таких как система Биткойн, транзакции предназначены для денежного обращения между всеми пользователями системы.

Каждая транзакция состоит из нескольких частей. При этом транзакция может быть простой или сложной в зависимости от того, из каких средств складывается итоговая сумма перевода и для кого предназначен платеж (или платежи).

Кроме того, для криптовалют, таких как биткойн, характерно свойство появления новых денег с каждым новообразованным блоком в цепочке. Каждый раз, когда майнер находит новый блок, он получает вознаграждение в виде какого-то количества биткойнов.

Первоначально вознаграждение за добытый блок составляло 50 BTC. Однако в систему Биткойн введена функция сокращения вознаграждения вдвое каждые 4 года. Данная процедура называется халвинг (от англ. half – половина). Таким образом, в 2012 г. произошло сокращение вознаграждения до 25 BTC, в 2016 г. – до 12,5 BTC, в 2020 г. – до 6,25 BTC.

Транзакция вознаграждения имеет структуру, которая отличается от других транзакций, при этом отличаются и правила формирования таких транзакций. В описании биткойн-системы такие транзакции называются Coinbase, то есть транзакции, которые вырабатываются самим ядром системы.

У Coinbase-транзакции фактически нет входов и есть только один выход. Данной транзакцией майнеру начисляются новые монеты системы. Транзакции Coinbase могут быть созданы только майнерами биткойнов, и они являются исключением из многих правил, определенных для обычных транзакций.

Первоначально Coinbase-транзакции содержали только сумму вознаграждения. Это характерно для всех первых блоков. При этом за саму Coinbase-транзакцию комиссия не взимается. Пример подобной транзакции для блока с высотой 10 показан на рис. 4.9.



Рисунок 4.9. Coinbase-транзакция в блоке биткойна с высотой 10

С тех пор, как в систему Биткойн ввели комиссию за майнинг транзакции, Coinbase-транзакция выполняет майнеру перевод, величина которого соответствует сумме вознаграждения за добытый блок и сумме всех комиссий за транзакции, помещенные в блок. Комиссия за саму Coinbase-транзакцию по-прежнему не взимается. Пример такой транзакции для блока с высотой 300 000 показан на рис. 4.10.



Рисунок 4.10. Coinbase-транзакция в блоке биткойна с высотой 300 000

Все остальные транзакции в биткойн-системе обязательно имеют как минимум один вход (показывает, откуда у пользователя появились деньги) и один выход (показывает, куда пользователь эти деньги тратит). Основная структура транзакции в биткойн-системе представлена на рис. 4.11.

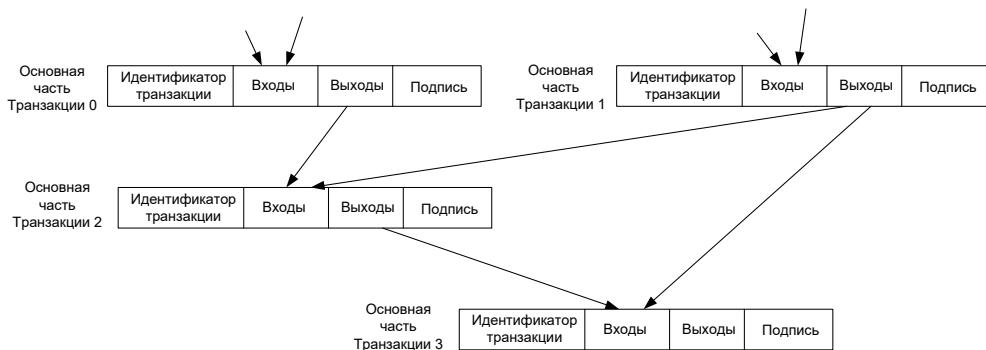


Рисунок 4.11. Связь входов и выходов в транзакциях

Если мы посмотрим на транзакцию через любой интернет-обозреватель (например, Blockchain Explorer), то сможем увидеть все входы и выходы транзакции. Рассмотрим для примера одну из первых транзакций в блоке с высотой 718 300 (рис. 4.12). Эта транзакция была создана 12 января 2022 года, содержит 14 входов и 2 выхода. Причем можем заметить, что все входы получены с одного адреса и имеют примерно равнозначное значение. Если внимательно посчитать, то можно увидеть, что сумма всех входов соответствует сумме выходов и комиссии.

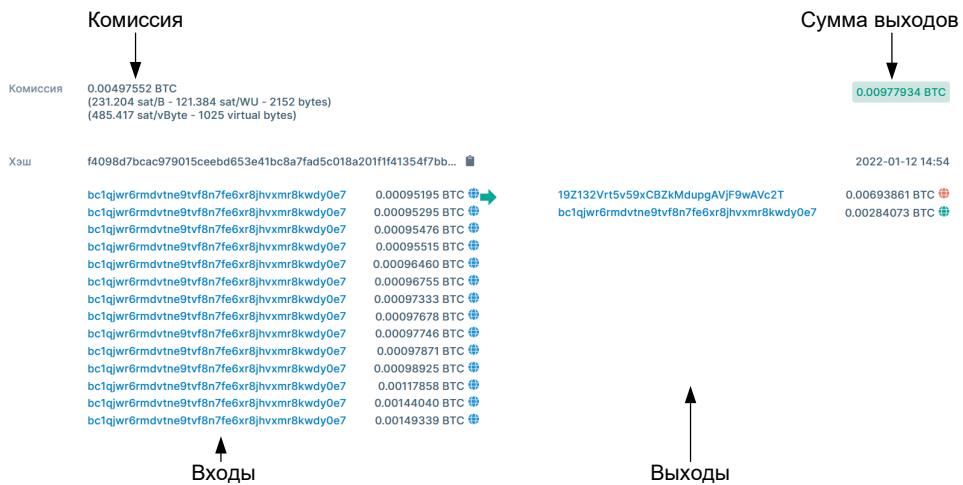


Рисунок 4.12. Входы и выходы транзакции в блоке биткоина с высотой 718 300

Каждый вход и выход можно рассмотреть более детально. При этом можно найти отметку о том, был уже использован выход транзакции в качестве входа для другой транзакции или еще нет (рис. 4.13).

Выходы ①

Индекс	0	Подробности	Израсходованные
--------	---	-------------	-----------------

Адрес	19Z132Vrt5v59xCBZkMd...	Стоимость	0.00693861 BTC
-------	-------------------------	-----------	----------------

Pkscript	OP_DUP OP_HASH160 <code>5dce58a0d0d0e3be1fa6d73519360bcab...0901f</code> OP_EQUALVERIFY OP_CHECKSIG
----------	---

Индекс	1	Подробности	Неизрасходованные
--------	---	-------------	-------------------

Адрес	bc1qjw...	Стоимость	0.00284073 BTC
-------	-----------	-----------	----------------

Pkscript	OP_O <code>9387a1edac5cf255b1279f939d186795d86d8cf6</code>
----------	---

Рисунок 4.13. Выходы транзакции в блоке биткоина с высотой 718 300

Каждый вход тратит биткойны или доли биткойна (сатоши), которые были получены в предыдущем выходе. Каждый выход может быть использован только один раз. Для неиспользованных выходов устанавливается статус неизрасходованной транзакции (UTXO). Так, на рис. 4.13 выход с индексом 1 имеет статус UTXO.

Когда вы проверяете свой баланс в биткойн-кошельке и видите, например, значение 1000 биткойнов, то это на самом деле означает, что сумма всех ваших неизрасходованных выходов транзакций составляет 1000 биткойнов.

Скриптовый язык для обработки транзакций

Для организации правильной работы по проверке и использованию транзакций используется специальный стековый язык Script (скрипт). С помощью простых команд этого языка выстраиваются инструкции, в соответствии с которыми необходимо формировать транзакцию и проверять валидность транзакции.

Для начала рассмотрим основные команды, которые используются в языке Script (табл. 4.2). В таблице приведен далеко не полный перечень команд для языка Script. Однако нам будет достаточно знать данные команды, для того чтобы разобраться с правилами построения транзакций для биткойн-системы. С полным перечнем команд для языка Script можно ознакомиться в [222].

Таблица 4.2. Основные команды языка Script

Команда	Вход	Выход	Описание
OP_0	Нет	Пустое значение	В стек помещается пустой массив байтов
N/A (определяется по коду операции)	Нет	Данные	Следующие за этой командой байты данных (от 1 до 75 в зависимости от кода операции) помещаются в стек
OP_1	Нет	1	В стек помещается значение 1
OP_2 – OP_16	Нет	2–16	В стек помещается одно из значений от 2 до 16
OP_IF	<выражение> IF [утверждение]		Если верхнее значение стека не ложно, то утверждение выполняется; верхнее значение стека удаляется
OP_VERIFY	Истина или Ложь	Ничего или ошибка выполнения	Если вершина стека соответствует утверждению Ложь, то помечает транзакцию как поврежденную; верхнее значение стека удаляется
OP_RETURN	Нет	Ошибка выполнения	Помечает транзакцию как недействительную; начиная с версии 0.9 ядра системы Биткойн стандартным способом добавления дополнительных данных к транзакции является добавление вывода с нулевым значением; при этом записывается операция OP_RETURN, за которой следуют данные

Окончание табл. 4.2

Команда	Вход	Выход	Описание
OP_DUP	x	$x\ x$	Дублирует верхнее значение стека
OP_EQUAL	x_1 x_2	Истина или Ложь	Возвращает 1, если входы точно равны, и 0 в противном случае
OP_EQUALVERIFY	x_1 x_2	Ничего или ошибка выполнения	То же, что и OP_EQUAL, только после этого запускается операция OP_VERIFY
OP_SHA256	x	Hash(x)	Вход хешируется с использованием алгоритма SHA-256
OP_HASH160	x	Hash(x)	Вход хешируется дважды: сначала с использованием алгоритма SHA-256, а затем с использованием алгоритма RIPEMD-160
OP_CHECKSIG	sig $pubkey$	Истина или Ложь	Все содержимое транзакции хешируется; подпись sig должна пройти верификацию для полученного хеша и открытого ключа $pubkey$; если проверка пройдена успешно, то возвращается 1, иначе возвращается 0
OP_CHECKSIGVERIFY	sig $pubkey$	Истина или Ложь	То же, что и OP_CHECKSIG, только после этого запускается операция OP_VERIFY
OP_CHECKMULTISIG	X sig_1 sig_2 \dots sig_x $pubkey_1$ $pubkey_2$ \dots $pubkey_x$	Истина или Ложь	Сравнивает первую подпись с каждым открытым ключом в соответствии с алгоритмом проверки подписи ECDSA; затем вторую и т. д.; процесс повторяется до тех пор, пока не будут проверены все подписи или пока не останется открытых ключей; если все подписи действительны, возвращается 1, иначе 0

Команды, записанные друг за другом, будут помещаться в стек (и обрабатываться) друг за другом. Так, например, запись «COM1 COM2» будет означать, что в стек будет сначала помещена команда COM1, а затем команда COM2. Таким образом команда COM1 будет находиться на дне стека, а команда COM2 – на его вершине.

Виды транзакций в системе Биткойн

В каждой транзакции записывается четырехбайтовый номер версии транзакции. Данный номер применяется для определения того набора правил, который необходимо использовать для ее проверки. Это позволяет разработчикам создавать новые правила для будущих транзакций без аннулирования предыдущих транзакций.

После того как в ранних версиях системы Биткойн было обнаружено несколько опасных ошибок, в систему был добавлен специальный тест. Тест про-

веряет синтаксис транзакции, правильность построения и соответствие всем установленным шаблонам. Этот тест называется *IsStandard*, и транзакции, которые проходят его, называются стандартными транзакциями.

И наоборот, нестандартные транзакции – это те транзакции, которые не прошли проверку и могут быть приняты узлами, не использующими настройки основного ядра системы Биткойн (Bitcoin Core) по умолчанию. Если они включены в блоки, значит, они избежали проверки *IsStandard* и будут обработаны.

Задачи, которые решает стандартный тест транзакций, состоят в следующем:

- предотвращение атак на систему Биткойн, связанных с генерацией и распространением вредоносных транзакций;
- запрет создания пользователями таких транзакций, которые в будущем могут усложнить добавление новых функций в состав транзакций.

Например, как было описано выше, каждая транзакция включает в свой состав номер версии. При этом если пользователи начнут произвольно изменять номер версии, она станет бесполезной в качестве инструмента для внедрения обратно несовместимых функций.

Каждая транзакция содержит в своем составе входы и выходы (как минимум один выход). При этом выходы содержат специальное поле, которое называется скриптом открытого ключа (*Pkscript*). Скрипт открытого ключа содержит набор инструкций на языке *Script* (из тех, что приведены в табл. 4.2), по которым необходимо будет выполнить проверку валидности транзакции, когда кто-нибудь захочет использовать эту транзакцию в качестве входа.

Выходы формируются из входов. Поэтому выход содержит скрипт открытого ключа (*Pkscript*), который был ему задан при формировании траты, а также содержит специальное поле скрипта подписи (*Sigscript*). Скрипт подписи содержит набор параметров, которые необходимо использовать для выполнения проверки в соответствии со скриптом открытого ключа.

Начиная с версии ядра 0.9 (Bitcoin Core 0.9), которая появилась в 2014 году, можно выделить следующие стандартные транзакции:

- транзакция на основе платы за хеш открытого ключа (*P2PKH*);
- транзакция на основе платы за хеш скрипта (*P2SH*);
- транзакция на основе коллективной подписи (*Multisig*);
- транзакция на основе использования только открытого ключа (*PubKey*);
- транзакция с нулевыми данными.

Транзакция на основе платы за хеш открытого ключа

Схема, при которой выход одной транзакции передает свои деньги скрипту открытого ключа, а затем вход другой транзакции может потратить эти деньги с использованием правильных значений в скрипте подписи, называется Pay-To-Public-Key-Hash (*P2PKH*), что дословно можно перевести как «плата за хеш открытого ключа». Рассмотрим эту схему более подробно.

На рис. 4.14 представлены фрагменты двух транзакций, при этом часть полей, не влияющая на понимание концепции образования транзакции, опущена.

Транзакция может содержать несколько входов и выходов. У каждого входа и выхода есть свой индекс. На рис. 4.14 показано, что в транзакциях будет представлено *m* входов и *n* выходов.

Выход всегда содержит некоторую сумму, которую пользователь перечисляет некоему адресату. Фактически в качестве такого адресата выступает скрипт открытого ключа (Pkscript), который будет описан далее. Сумму, назначенную в данном выходе, сможет потратить тот, кто пройдет аутентификацию со стороны скрипта открытого ключа.

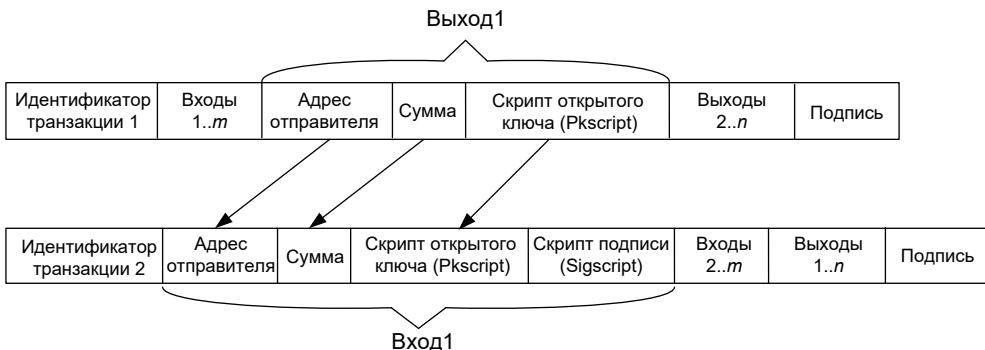


Рисунок 4.14. Преобразование выхода во вход

Фактически скрипт открытого ключа представляет собой набор условий, которые должны быть выполнены, а также обеспечивает автоматическую проверку соответствия данным условиям предоставляемых ему данных.

Формируемый вход, помимо всех основных полей выхода, из которого он образован, также содержит в своем составе скрипт подписи (Sigscript). Скрипт подписи содержит те данные, которые проверяются механизмами скрипта открытого ключа. Смысл заключается в том, что потратить данный выход сможет тот, кто может доказать, что знает закрытый ключ, соответствующий хешированному открытому ключу.

По правилам биткойн-системы транзакции не хранят открытый ключ пользователя в открытом виде. Это сделано в том числе для того, чтобы у злоумышленников было как можно меньше шансов восстановить закрытый ключ, от которого зависят активы пользователя. Однако транзакция содержит слепок открытого ключа, образованный с использованием хеш-кода.

Для того чтобы Алиса смогла перевести Бобу деньги, Боб должен быть зарегистрирован в системе. Регистрация в системе подразумевает, что для Боба должна быть сгенерирована пара ключей: закрытый и открытый.

В системе Биткойн используется алгоритм электронной подписи на основе эллиптических кривых (ECDSA) с кривой secp256k1 [90] (данний алгоритм подробно описан в главе 2). Закрытый ключ для ECDSA с кривой secp256k1 представляет собой 256 бит случайных данных. Копия этих данных детерминированно преобразуется в открытый ключ. В силу того, что данное преобразование можно надежно повторить позже, открытый ключ самому пользователю хранить не обязательно, достаточно знать закрытый ключ.

Выработанный открытый ключ хешируется. Полученный в результате хеш-код сокращает длину ключа, а также скрывает само значение открытого ключа. Уменьшение длины ключа упрощает ввод его значения вручную.

Пользователь, который вырабатывает хеш-код, всегда может повторить данный процесс, зная исходные данные и алгоритм хеширования. Поэтому ему не обязательно хранить вычисленный хеш-код.

Итак, схема работает следующим образом. Алиса хочет перевести Бобу некоторую сумму. На компьютере Боба хранится его закрытый ключ. На компьютере Боба из закрытого ключа вырабатывается открытый ключ, а затем его хеш-код. Именно этот хеш-код Боб должен сообщить Алисе, для того чтобы она смогла перевести Бобу деньги.

Используя полученный хеш-код открытого ключа, Алиса формирует скрипт открытого ключа для создаваемой транзакции, помещая в него полученный хеш-код открытого ключа (см. рис. 4.15).

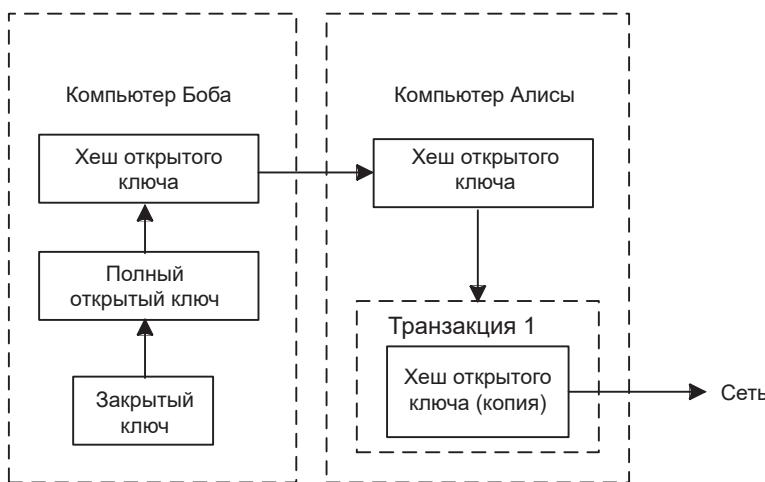


Рисунок 4.15. Создание P2PKH для получения платежа

Обычно хеш-код открытого ключа записывается в кодировке биткойн-адресов, которые представляют собой строки в кодировке base58, содержащие номер версии адреса, хеш-код и контрольную сумму для выявления опечаток.

Адрес может быть передан через любой носитель, и его можно дополнитель- но закодировать в другой формат, например QR-код, содержащий сообщение вида «`<<bitcoin:> URI`», где URI (Uniform Resource Identifier) – это унифицирован- ный идентификатор (адрес) ресурса.

Если адрес был дополнительно закодирован, то после его получения Али- са должна декодировать его обратно в стандартный хеш-код. После этого она может создать транзакцию для отправки денег Бобу. Для этого она должна соз- дать стандартный выход транзакции P2PKH (см. рис. 4.15), содержащий скрипт открытого ключа.

Алиса формирует транзакцию и отправляет ее в сеть. После того как транзакция будет замайнена, она добавляется в цепочку блоков. Сеть классифицирует выходы транзакции как неизрасходованный выход транзакции (UTXO), а программное обеспечение кошелька Боба отображает его как баланс, который можно потратить.

Когда через некоторое время Боб решит потратить свои неизрасходованные выходы, он должен будет создать новый вход (по принципу, показанному на

рис. 4.14) и в этом входе указать ссылку на данные выхода транзакции Алисы. Затем Боб должен создать скрипт подписи, то есть набор тех данных, которые пройдут все проверки, заданные Алисой в скрипте открытого ключа (рис. 4.16).

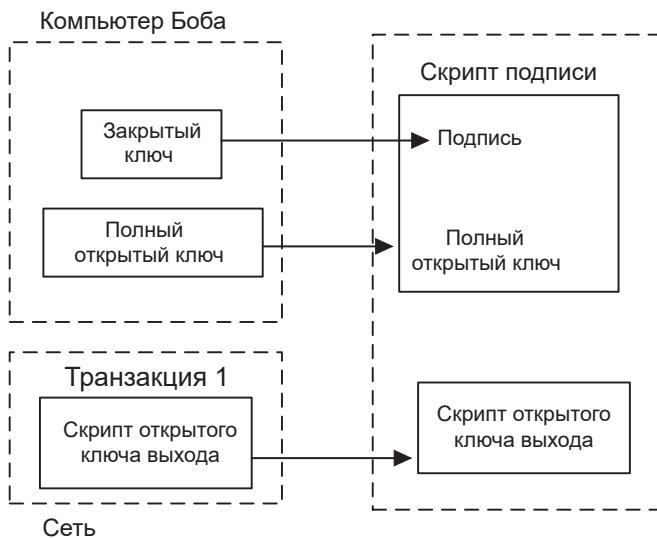


Рисунок 4.16. Разблокирование выхода Р2РКН для следующей траты

Фактически Боб должен поместить в скрипт подписи два компонента: свою подпись, сделанную для данных транзакции закрытым ключом Боба в соответствии с алгоритмом ECDSA, а также свой полный открытый ключ. При этом хеш-код открытого ключа из Транзакции 1 (см. рис. 4.14) также перепишется из соответствующего выхода Транзакции 1.

Проверка данных параметров впоследствии осуществляется скриптом открытого ключа и состоит в следующем:

1. Из полного открытого ключа Боба будет получен его хеш-код. Он должен совпасть с хеш-кодом открытого ключа, помещенным в скрипт подписи. Это подтвердит тот факт, что деньги действительно предназначены тому пользователю, которого указала Алиса.
2. Будет выполнена проверка подписи с использованием алгоритма ECDSA и кривой secp256k1, чтобы убедиться в том, что Боб действительно владеет закрытым ключом, из которого был выработан данный открытый ключ.

Проверка подписи Боба не просто доказывает, что Боб знает свой закрытый ключ, но также не позволяет кому бы то ни было внести изменения в состав самой транзакции. Поэтому транзакцию можно спокойно распространять в одноранговой сети, не опасаясь подделки.

При создании подписи подписывается вся транзакция, за исключением полного открытого ключа и самой подписи транзакции. Эти данные (полный открытый ключ и подпись транзакции) помещаются в скрипт подписи, после чего Боб отправляет сформированную транзакцию майнерам через одноранговую сеть.

Каждый одноранговый узел и майнер независимо друг от друга проверяют транзакцию, прежде чем передавать ее дальше или пытаться включить ее в новый блок транзакций.

P2PKH – это наиболее распространенная форма скрипта открытого ключа, используемая для отправки транзакции на один или несколько адресов системы Биткойн. Для формирования скрипта открытого ключа и скрипта подписи используется стандартный набор инструкций в соответствии с табл. 4.3.

Таблица 4.3. Основные скрипты сценария «Плата за хеш открытого ключа»

Скрипт открытого ключа	<code>OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG</code>
Скрипт подписи	<code><Sig><PubKey></code>

Скрипт подписи того пользователя, который намеревается потратить деньги, проверяется, и его данные (подпись с использованием алгоритма ECDSA с кривой secp256k1 и полный открытый ключ) записываются перед началом скрипта открытого ключа. В результате в соответствии с табл. 4.3 образуется следующая последовательность:

`<Sig><PubKey> OP_DUP OP_HASH160 <PubkeyHash5> OP_EQUALVERIFY OP_CHECKSIG`

Чтобы проверить, действительна ли транзакция, скрипт подписи и скрипт открытого ключа проверяют последовательно все элементы, начиная со скрипта подписи Боба и заканчивая скриптом открытого ключа Алисы, последовательно помещая данные в стек (рис. 4.17).

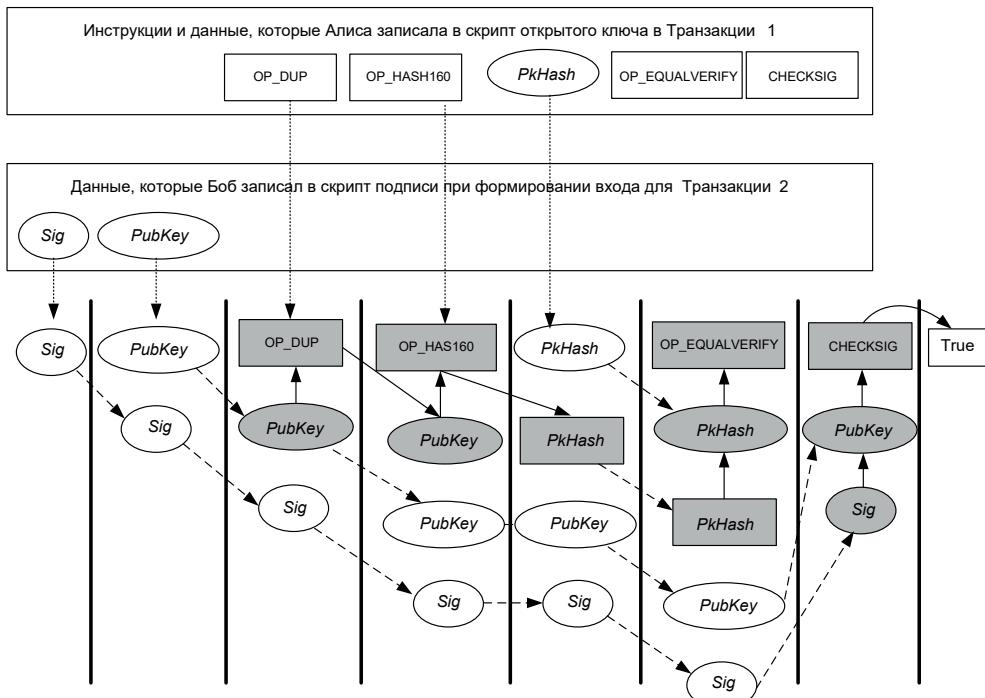


Рисунок 4.17. Схема последовательной проверки всех элементов

Подпись, которую Боб сформировал для входа Транзакции 2 (*Sig*), а также полный открытый ключ Боба (*PubKey*) просто помещаются в стек и используются для проверки (шаги 1 и 2 на рис. 4.17).

Скрипт открытого ключа Алисы выполняет операцию OP_DUP (шаг 3 на рис. 4.17). OP_DUP помещает в стек копию данных, находящихся в данный момент наверху, – в этом случае создается копия открытого ключа, предоставленного Бобом.

Следующая выполняемая операция – OP_HASH160 (шаг 4 на рис. 4.17). Эта операция вырабатывает хеш-код из тех данных, которые в настоящий момент находятся в вершине стека, – в данном случае это открытый ключ Боба. Таким образом создается хеш-код для открытого ключа Боба (шаг 5 на рис. 4.17), который Боб сообщил Алисе для последующей проверки.

Получается, что в этот момент в верхней части стека находятся две копии хеш-кода ключа Боба (одна была записана в стек открытого ключа Алисы, а вторая выработалась с помощью инструкции OP_HASH160).

Следующая инструкция в скрипте открытого ключа Алисы – это инструкция OP_EQUALVERIFY (шаг 6 на рис. 4.17). Фактически данная инструкция состоит из двух шагов: проверки OP_EQUAL и подтверждения OP_VERIFY.

На первом шаге OP_EQUAL проверяет два значения в вершине стека (сравнивает хеш-коды открытого ключа Боба). После этого сравниваемые хеш-коды удаляются из вершины стека, а вместо них записывается 1, если сравнение выполнено успешно, и 0 в противоположном случае.

На втором шаге OP_VERIFY выполняется проверка значения, записанного в вершину стека. Если в вершине находится значение 0, то дальнейшая проверка транзакции не проводится и алгоритм прекращает свою работу. Иначе, если значение в вершине стека равно 1, оно выталкивается из стека и проверка продолжается.

Следующая инструкция в скрипте открытого ключа Алисы – это OP_CHECKSIG (шаг 7 на рис. 4.17). С помощью данной инструкции проверяется подпись, предоставленная Бобом, с использованием открытого ключа Боба (который на предыдущем шаге прошел аутентификацию).

Если проверка подписи с использованием открытого ключа выполняется успешно, то в вершину стека помещается значение 1 (true), иначе помещается значение 0 (false) (шаг 8 на рис. 4.17). Именно это значение определяет в конечном итоге, признается транзакция действительной или нет.

На рис. 4.18 представлен пример перехода выхода одной транзакции во вход другой транзакции. В данном примере выход № 15 транзакции 27e9943d2197d a7578a24a46272700b54d93175e1220b72207665b909a666a57 преобразуется в нулевой вход транзакции c1b7b41cc7bc40fe528a08528ae150e2fdab6aa8501dd100a c64b194241cf9b2.

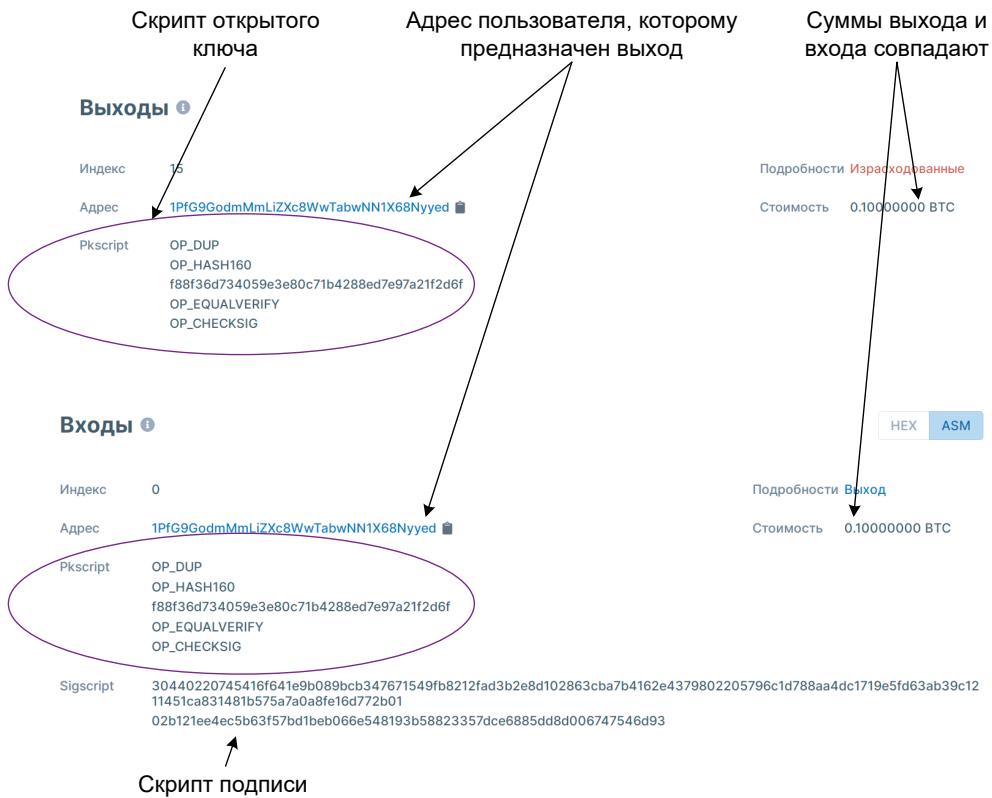


Рисунок 4.18. Пример преобразования выхода во вход по принципу P2PKH

Транзакция на основе платы за хеш скрипта

Скрипты открытых ключей создаются отправителями транзакций, которых мало интересует, как работает этот скрипт. Получатели же транзакций заинтересованы в правильности выполнения всех проверок, и поэтому они могут попросить отправителя использовать определенный скрипт открытого ключа. Однако ссылки на пользовательские сценарии открытых ключей менее удобны, чем короткие адреса биткойнов, и не было стандартного способа передачи их между программами до широко распространенной реализации ныне устаревшего платежного протокола BIP70 [57].

Чтобы решить эти проблемы, в 2012 году были созданы транзакции вида Pay-To-Script-Hash (P2SH), что дословно можно перевести как «Плата за хеш скрипта». Такие транзакции позволяют отправителю создать скрипт открытого ключа, который в свою очередь содержит второй скрипт, который называется скриптом погашения (Redeem Script).

Принцип формирования транзакции вида P2SH (рис. 4.19) очень похож на формирование P2PKH. Боб создает скрипт погашения с любым сценарием проверки, который он хочет, хеширует скрипт погашения и предоставляет Алисе хеш-код скрипта погашения. Алиса создает выход в стиле P2SH, содержащий хеш-код скрипта погашения Боба.

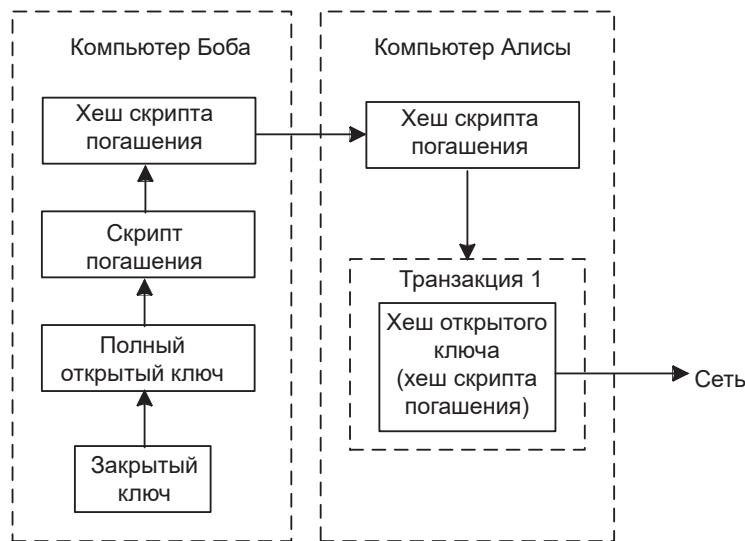


Рисунок 4.19. Создание P2SH для получения платежа

Когда Боб хочет потратить деньги из выхода Алисы, он предоставляет свою подпись вместе с полным скриптом погашения в скрипте подписи (рис. 4.20). В остальном же работа по проверке скрипта погашения аналогична работе по проверке скрипта открытого ключа, и Боб может потратить деньги, если после проверки скрипт возвращает значение true.

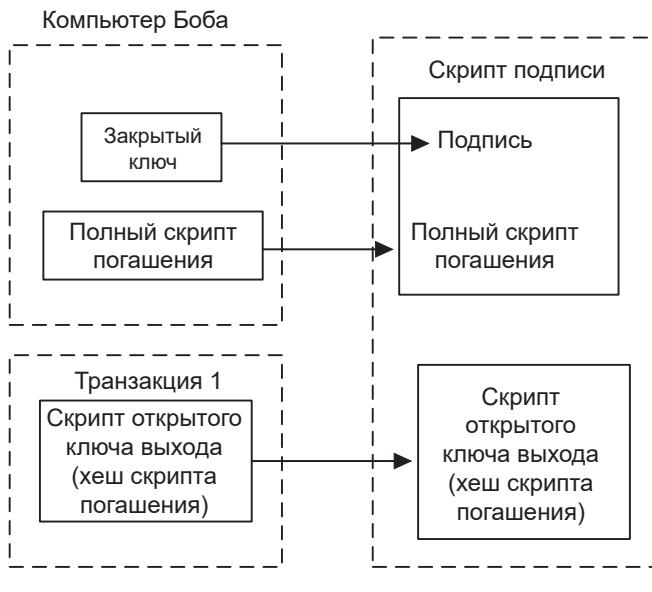


Рисунок 4.20. Разблокирование выхода P2SH для следующей траты

Хеш-код скрипта погашения имеет те же свойства, что и хеш-код скрипта открытого ключа, поэтому его можно преобразовать в стандартный формат адреса системы Биткойн с одним небольшим изменением, чтобы отличить его от стандартного адреса. Таким образом, формирование адреса для транзакции P2SH выполняется так же просто, как и для транзакции P2PKH.

Хеш-код тоже скрывает любые открытые ключи, используемые в скрипте погашения, поэтому транзакции P2SH так же безопасны, как и транзакции P2PKH.

P2SH используется для обработки транзакции в соответствии со скриптом погашения.

Каждый из стандартных скриптов открытого ключа может использоваться как скрипт погашения P2SH, за исключением самого P2SH. Начиная с версии ядра 0.9.2 (Bitcoin Core 0.9.2) транзакции P2SH могут содержать любой действительный код, записанный в скрипт погашения, что делает механизм P2SH гораздо более гибким и позволяет экспериментировать со многими новыми и сложными типами транзакций.

Чаще всего механизм P2SH используется для стандартного сценария открытого ключа с несколькими подписями, а вторым по распространенности является протокол для открытых активов (Open Assets). P2SH также обеспечивает удобный метод хранения текста в цепочке блоков, поскольку он позволяет хранить до 1,5 КБ текстовых данных.

Для формирования скрипта открытого ключа и скрипта погашения используется стандартный набор инструкций в соответствии с табл. 4.4.

Таблица 4.4. Основные скрипты сценария «Плата за хеш скрипта»

Скрипт открытого ключа	OP_HASH160 <Hash160 (redeemScript)> OP_EQUAL
Скрипт подписи	<sig> [sig] [sig ...] <redeemScript>

Такая организация скрипта открытого ключа отлично подходит для старых узлов в том случае, если хеш-код скрипта соответствует скрипту погашения. Однако после активации softfork новых узлов будут выполнять дополнительную проверку скрипта погашения. Они извлекут скрипт погашения из скрипта подписи, декодируют его и применят извлеченные инструкции для проверки оставшихся элементов стека (часть <sig> [sig] [sig ...]).

Следовательно, чтобы получатель мог использовать транзакцию P2SH, отправитель должен предоставить действительную подпись или ответ в дополнение к правильному скрипту погашения. Этот последний шаг аналогичен шагу проверки в сценариях P2PKH или коллективной подписи (см. далее), в которых начальная часть скрипта подписи (<sig> [sig] [sig ...]) действует как скрипт подписи, а скрипт погашения действует как скрипт открытого ключа.

На рис. 4.21 представлен пример перехода выхода одной транзакции во вход другой транзакции по принципу P2SH, где выход № 9 транзакции 8998bb9c154000e6d1ac9b4452550fe35b287c2cbe618d1cea70dc1555270078 преобразуется во вход № 2 транзакции 41eb724cbc90b6d918513befef0a42149bb17e0e7422ca87d72615f0b6437f3c.

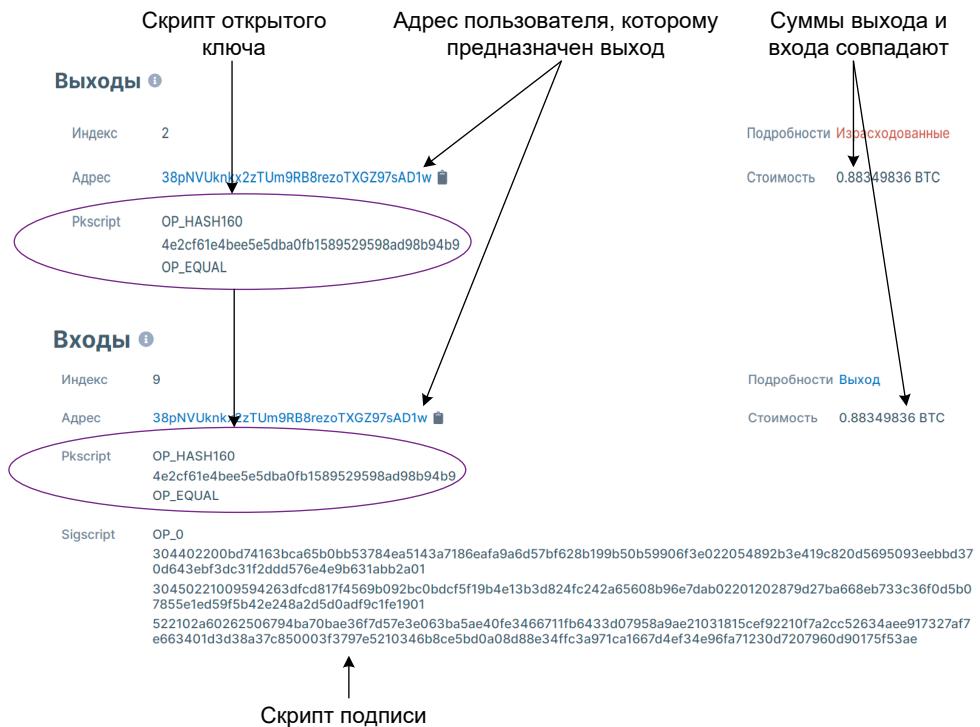


Рисунок 4.21. Пример преобразования выхода во вход по принципу P2SH

Транзакция на основе коллективной подписи

Хотя механизм коллективной подписи сейчас обычно используется для транзакций с несколькими подписями, этот базовый скрипт может использоваться для запроса нескольких подписей, перед тем как можно будет потратить неизрасходованный выход UTXO.

В скрипте с несколькими подписями используется формула вида m -из- n , где m – это минимальное количество подписей, которое должно соответствовать открытому ключу, а n – это количество предоставленных открытых ключей. И m , и n в формате скрипта должны быть представлены кодами операций с OP_1 по OP_16 в соответствии с желаемым номером.

Из-за ошибки в исходной реализации системы Биткойн, которую нельзя исправить из-за проблем совместимости (они неизбежно возникнут в случае исправления), операция «OP_CHECKMULTISIG» забирает из стека на одно значение больше, чем указанное значение m , поэтому список подписей в скрипте подписи должен начинаться с предварительного значения (OP_0), которое будет указано, но не будет использовано.

Скрипт подписи должен предоставлять подписи в том же порядке, в каком соответствующие открытые ключи появляются в скрипте открытого ключа или скрипте погашения (табл. 4.5).

Таблица 4.5. Основные скрипты сценария «Коллективная подпись»

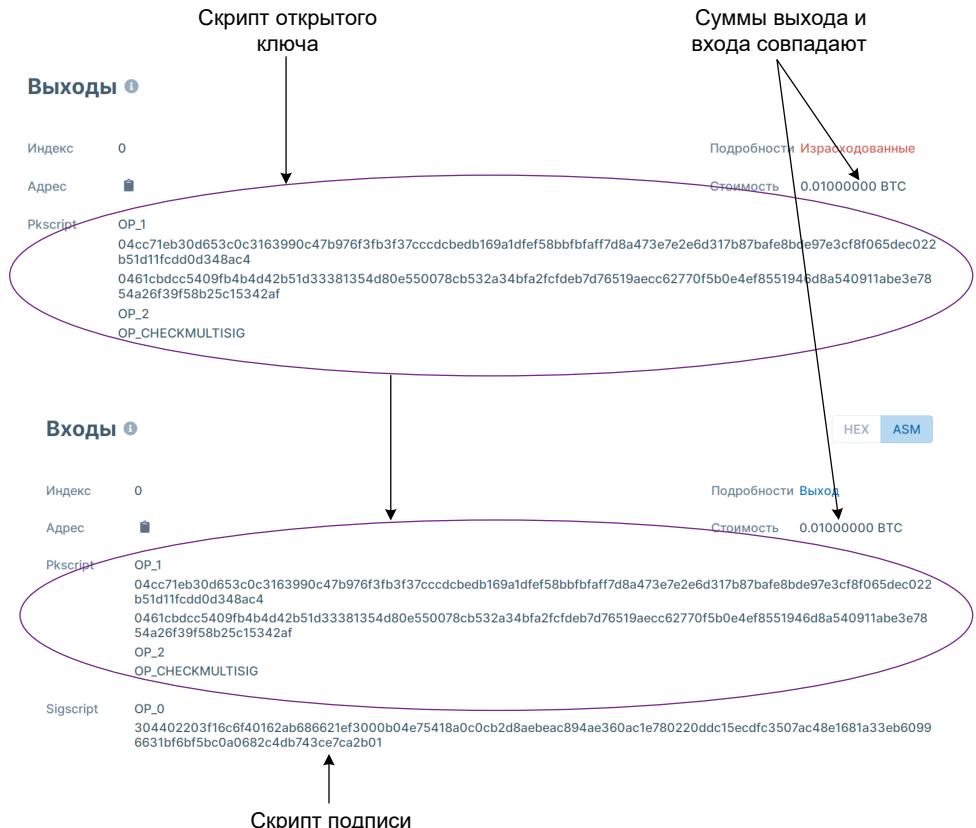
Скрипт открытого ключа	$<m><A \text{ pubkey}>[B \text{ pubkey}][C \text{ pubkey} \dots]<n>$ OP_CHECKMULTISIG
Скрипт подписи	OP_0 <Asig> [Bsig] [Csig ...]

Фактически это получается не отдельный тип транзакции, а использование механизма P2SH (два раза из трех возможных), как показано в табл. 4.6.

Таблица 4.6. Альтернативное представление сценария «Коллективная подпись»

Скрипт открытого ключа	OP_HASH160 <Hash160 (redeemScript)> OP_EQUAL
Скрипт активации	<OP_2><A pubkey><B pubkey><C pubkey><OP_3> OP_CHECKMULTISIG
Скрипт подписи	OP_0 <A sig><C sig><redeemScript>

Пример транзакции с мультиподписью приведен на рис. 4.22. Здесь показан выход № 0 для транзакции 60a20bd93aa49ab4b28d514ec10b06e1829ce6818ec06 cd3aab013ebcdc4bb1, который был потрачен в транзакции 23b397edcccd3740a 74adb603c9756370fafcde9bcc4483eb271ecad09a94dd63, образуя единственный вход данной транзакции.

**Рисунок 4.22.** Пример преобразования выхода во вход по принципу мультиподписи

Транзакция на основе использования только открытого ключа

Способ, когда при построении транзакции используется только открытый ключ, является упрощенной формой скрипта открытого ключа P2PKH. Но данный подход не является безопасным (в отличие от P2PKH), поэтому он больше не используется в новых транзакциях.

Для формирования скрипта открытого ключа и скрипта подписи используется стандартный набор инструкций в соответствии с табл. 4.7.

Таблица 4.7. Основные скрипты сценария «Только открытый ключ»

Скрипт открытого ключа	<pubkey> OP_CHECKSIG
Скрипт подписи	<sig>

На рис. 4.23 приведен пример выхода, преобразованного во вход по принципу «Только открытый ключ» для одних из ранних транзакций (блок с высотой 170, созданный в январе 2009 года). На рис. 4.23 показан выход № 0 для транзакции ea44e97271691990157559d0bdd9959e02790c34db6c006d779e82fa5ae e708e, который был потрачен в качестве входа № 0 транзакции f4184fc596403b 9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16, образуя единственный вход данной транзакции.

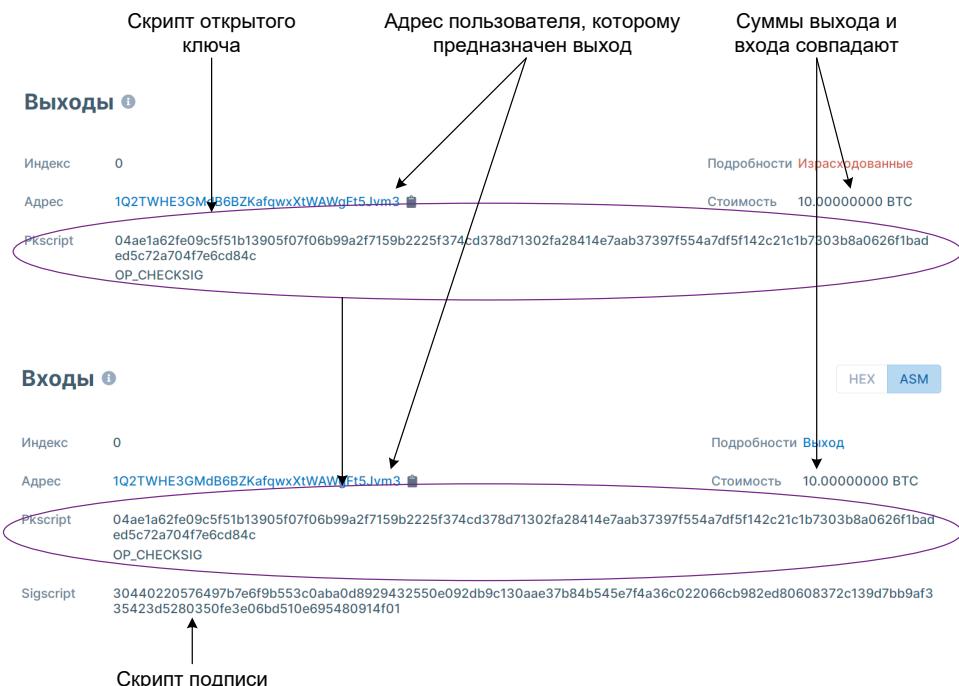


Рисунок 4.23. Пример транзакции по принципу «Только открытый ключ»

Транзакция с нулевыми данными

Не получил широкой известности тот факт, что в системе Биткойн сегодня можно совершать не только денежные платежи, но и отправлять какие-либо данные вместо криптовалюты. Такие транзакции называются транзакциями с нулевыми данными (NullData).

В системе Биткойн транзакции с нулевыми данными появились с обновлением системы в марте 2014 года и с переходом к ядру системы версии 0.9.0 (Bitcoin Core 0.9.0) [78]. Чтобы сохранить данные в блокчейне, вы просто отправляете транзакцию, но вместо того, чтобы отправлять ее на другой биткойн-адрес, вы отправляете транзакцию с нулевыми данными с выходом нулевого значения. Получается, что один из выходов содержит не сумму и адрес, а сообщение и нулевую сумму.

Для обрабатывающих узлов нет необходимости сохранять такие транзакции в списке транзакций с непотраченными выходами (UTXO). Тем не менее разработчики системы обращают внимание пользователей на то, что предпочтительнее хранить лишние данные вне блокчейна, чтобы не раздувать его.

Правила консенсуса позволяют добавлять в блок транзакции с ненулевыми данными до максимально допустимого размера сценария открытого ключа (10 000 байт), при условии что все транзакции соответствуют всем остальным правилам консенсуса (таким как, например, отсутствие отправки данных размером более 520 байт).

Тем не менее для версий ядра Bitcoin Core 0.9.x-0.10.x по умолчанию будут формироваться и обрабатываться транзакции, которые содержат нулевые данные размером до 40 байт и только один выход, в котором указана сумма в 0 сатоши (табл. 4.8).

Таблица 4.8. Основные скрипты сценария «Транзакция с нулевыми данными»

Скрипт открытого ключа	OP_RETURN <от 0 до 40 байт данных>
Скрипт подписи	Транзакции с нулевыми данными не могут быть потрачены, поэтому скрипт подписи отсутствует

До того, как в ядро системы биткойн была введена проверка совпадения суммы входа и суммы выхода, транзакция с нулевыми данными позволяла потратить в качестве входов некоторую сумму биткойнов, но при этом не формировать выход. Фактически деньги оказываются замороженными. То есть они были потрачены, но больше никогда не могут быть использованы.

Так, например, транзакция eb31ca1a4cbd97c2770983164d7560d2d03276ae1aee26f12d7c2c6424252f29, созданная 29 марта 2013 года, имеет два входа на 0.075 BTC и 0.05 BTC соответственно, что на начало января 2022 года составляло примерно 400 000 рублей. При этом выход транзакции содержит скрипт открытого ключа OP_RETURN с суммой 0 BTC (рис. 4.24).

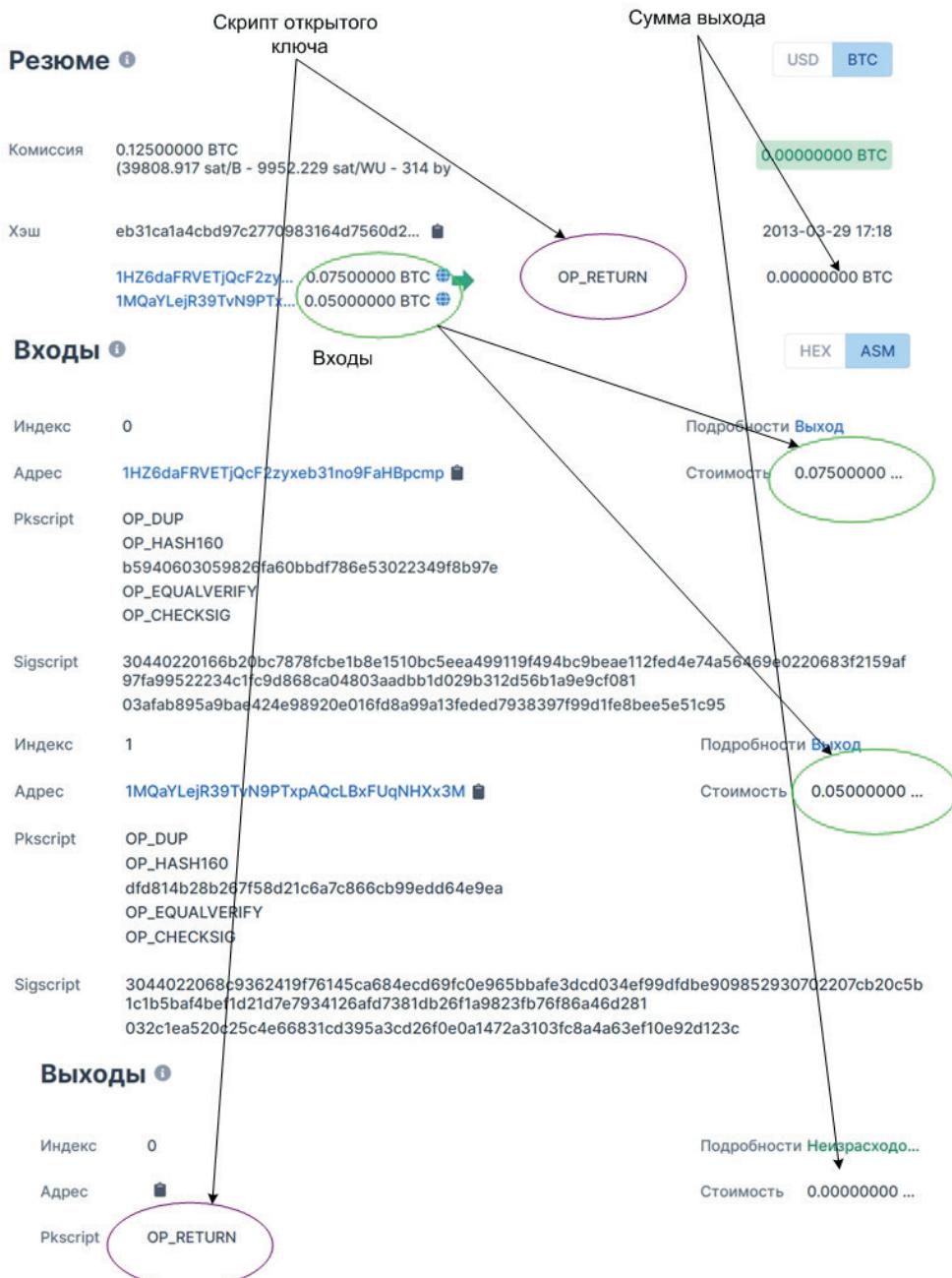


Рисунок 4.24. Пример транзакции с нулевыми выходами (заморозка биткойнов)

Еще одним ярким примером транзакции с нулевым выходом является транзакция, в которой после команды OP_RETURN записывается некоторое послание в формате UTF-8. Так, например, в транзакции 8bae12b5f4c088d940733dcd 1455efc6a3a69cf9340e17a981286d3778615684, созданной 30 июня 2014 года, выход № 0 не содержит денежного перевода. Зато он содержит послание, которое в соответствующей кодировке гласит «charley loves heidi», что значит «Чарли любит Хайди» (рис. 4.25).

Выходы	
Индекс	0
Адрес	[]
Pkscript	OP_RETURN 636861726c6579206c6f766573206865696469
Индекс	1
Адрес	1HnhWpkMHMjgt167kvgcPyurMmsCQ2WPgg []
Pkscript	OP_DUP OP_HASH160 b8268ce4d481413c4e848ff353cd16104291c45b OP_EQUALVERIFY OP_CHECKSIG

UTF-8 в строку

UTF-8

636861726c6579206c6f766573206865696469

Формат данных
Определить автоматически ▾

Результат
charley loves heidi

Рисунок 4.25. Пример транзакции с нулевыми выходами (любовное послание)

В версии ядра Bitcoin Core 0.11.x размерность транзакции по умолчанию была увеличена до 80 байт, при этом остальные правила остались прежними. В версии Bitcoin Core 0.12.0 размер транзакции по умолчанию был увеличен до 83 байт. При этом было снято ограничение на размерность одной транзакции. Главным оставалось условие не превышать общую размерность блока.

Параметр конфигурации --data carrier size для ядра Bitcoin Core позволит вам установить максимальное количество байтов в транзакциях с нулевыми данными, которые вы планируете формировать и отправлять. Пример по формированию таких транзакций с использованием специальной библиотеки Python Bitcoin Library [204] для языка Python можно найти в статье [239].

Нестандартные транзакции

Если в выходных данных транзакции содержится что-либо, кроме стандартного значения скрипта открытого ключа, то узел ядра системы Биткойн (Bitcoin Core) с настройками, заданными по умолчанию, не будет принимать, пересыпать и майнить такую транзакцию.

Если вы создадите скрипт погашения, захешируете его и будете использовать полученный хеш-код как выход для транзакции P2SH, блокчейн-сеть будет видеть только хеш-код и, как следствие, она будет считать выход действительным, независимо от того, что выявляет скрипт погашения.

Такой механизм позволяет производить оплату нестандартным скриптом, а с версии ядра Bitcoin Core 0.11 можно потратить почти все действующие скрипты погашения.

Исключение составляют скрипты, в которых используются неназначенные коды операций NOP; эти коды операций зарезервированы для будущих soft-форков. Транзакции с такими кодами операций могут быть обработаны и замайнены только узлами, которые не соответствуют стандартной политике для работы мемпула (пула памяти).

В исследовательской работе [77] авторы провели поиск нестандартных транзакций и определили десять их видов, характерных для системы Биткойн в разное время существования (табл. 4.9).

Таблица 4.9. Основные виды нестандартных транзакций

Вид транзакции	Скрипт открытого ключа	Год возникновения
Pay to Public Key Hash 0	OP_DUP OP_HASH160 0 OP_EQUALVERIFY OP_CHECKSIG	2011
P2PKH NOP	OP_DUP OP_HASH160 <HASHPUBLICKEY> OP_EQUALVERIFY OP_CHECKSIG OP_NOP	2011, 2014
OnlyHash	Только хеш	2011–2014
P2Pool Bug	OP_IFDUP OP_IF OP_2SWAP OP_VERIFY OP_2OVER OP_DEPTH	2012
CLTV	<DATA>OP_CHECKLOCKTIMEVERIFY OP_DROP	2012
OP_MIN OP_EQUAL	OP_MIN 3 OP_EQUAL	2012
Pay to Hash (P2H)	OP_HASH160 <HASH160 OF SOMETHING> OP_EQUALVERIFY	2012–2015
UnLocked (UL)	Пустой скрипт	2015
OP_RETURN ERROR	OP_RETURN ERROR	2016–2017
OP_2 OP_3 ERROR	OP_2 OP_3 ERROR	2017–2018

Так, например, в ноябре 2011 г. биржа Mt.Gox стала жертвой ошибки в своем программном обеспечении (ПО) [17], вследствие чего 2609.36304319 BTC было отправлено на адрес биржи с использованием Pay to Public Key Hash 0. Эти деньги остались навсегда замороженными в системе Биткойн. На январь 2022 года эта сумма составляет 8,5 миллиарда рублей.

Сайт Blockchain.com (с помощью которого, в частности, сделана большая часть рисунков в настоящей книге) на январь 2022 года не отображает адреса, для которых используются нестандартные транзакции, но просто выводит сообщение «Невозможно декодировать выходной адрес». Пример нестандартного выхода для транзакции 6d5088c138e2fbf4ea7a8c2cb1b57a76c4b0a5fab5f4c188696aad807a5ba6d8 показан на рис. 4.26.

Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 597 bytes)	7329.78709196 BTC
Хеш	6d5088c138e2fbf4ea7a8c2cb1b57a76c... ↗	2011-10-29 00:11
	16HhGwwLyugCYKv6Kt... 0.30553511 BTC ↗ 1KyzznvEnSJPe5zz... 7288.59790508 BTC ↗ 1FXkRcYtwDeKYerrT... 40.88365177 BTC ↗	Невозможно декодировать выходной ад... 1L4TuuUdk2ouC9X... 7283.96709196 BTC ↗

Выходы ⓘ

Индекс	0	Подробности Неизрасходо...
Адрес	s-272edf45031dd498e7b3ae89e11ff21b ↗	Стоимость 45.82000000...
Pkscript	OP_DUP OP_HASH160 OP_0 OP_EQUALVERIFY OP_CHECKSIG	

Рисунок 4.26. Пример нестандартной транзакции

Однако альтернативный ресурс blockchair.com отображает адреса с нестандартными транзакциями и позволил определить адрес биржи Mt.Gox, на который были выведены деньги. Адрес имеет нестандартный вид s-272edf45031dd498e7b3ae89e11ff21b (рис. 4.27).



Рисунок 4.27. Кошелек s-272edf45031dd498e7b3ae89e11ff21b

Кроме того, ресурс blockchair.com позволяет получить специальную выписку по кошельку, в которой видны все совершенные транзакции. По адресам данных транзакций можно проверить, что все они соответствуют нестандартному скрипту Pay to Public Key Hash 0. На рис. 4.28 приведен фрагмент выписки, полную выписку можно получить по ссылке [2].

 BLOCKCHAIR info@blockchair.com https://blockchair.com 03/01/2009 - 24/06/2022 (Part 1/1)				

WALLET STATEMENT	BITCOIN			

WALLET ADDRESS: s-272edf45031dd498e7b3ae89e11ff21b				
STATEMENT PERIOD: 03/01/2009 - 24/06/2022				

BTC BALANCE SUMMARY:				

НАЧАЛЬНЫЙ БАЛАНС (03/01/2009)	0.00 USD			
ВСЕГО ПОЛУЧЕНО	2,609.36304319 BTC			
ВСЕГО ОТПРАВЛЕНО	0.00000000 BTC			
ENDING BALANCE (24/06/2022)	54,937,529.51 USD			

HISTORY OF TRANSACTIONS: 03/01/2009 - 24/06/2022				

#	ВРЕМЯ	СУММА (BTC)	СУММА (USD)	TRANSACTION HASH
1	2011-10-28 20:11:28	Полученный 45.82000000	133.79	6d5088c138e2fbf4ea7a8c2cb1b57a6d8 4b0a5fab5f4c188696aad807a5ba6d8
2	2011-10-28 20:11:28	Полученный 100.00000000	292.00	07d33c8c74e945c50e45d3eaf4add7553 534154503a478cf6d48e1c617b3f9f3
3	2011-10-28 20:11:28	Полученный 98.00000000	286.16	6d39eeb2ae7f9d42b0569cf1009de4c9f 031450873bf2ec84ce795837482e7a6
4	2011-10-28 20:11:28	Полученный 200.00000000	584.00	2d00ef4895f20904d7d4c0bada17a8e9d 47d6c049cd2e5002f8914bfafa7f1d27b
5	2011-10-28 20:11:28	Полученный 100.00000000	292.00	15ad0894ab42a46eb04108fb8bd667865 66a74356d2103f077710733e0516c3a
6	2011-10-28 20:11:28	Полученный 497.00000000	1,451.24	03acfaf47d1e0b7674f1193237099d155 3d3d8a93ecc85c18c4bec37544fe386
7	2011-10-28 20:11:28	Полученный 100.00000000	292.00	3ab5f53978850413a273920bfc86f4278 d9c418272accddade736990d60bdd53

Рисунок 4.28. Выписка по кошельку s-272edf45031dd498e7b3ae89e11ff21b

Ресурс bitcoin-supply.com отображает как самые большие, так и последние потери в системе Биткойн. Здесь можно получить информацию о номере блока, в котором произошла утрата, идентификатор и сумму транзакции.

Здесь же стоит упомянуть о том, что в ранних версиях системы Биткойн не проводилось отслеживание создания двойных идентичных транзакций. Самы-

ми подверженными этой уязвимости оказались Coinbase-транзакции, по которым майнер получает вознаграждение за созданный блок.

Так как Coinbase-транзакция не содержит в своем составе ссылок на предыдущие выходы и формируется по одному и тому же принципу, то известно, по крайней мере, два случая, когда одинаковая транзакция была создана и начислена майнеру. Так, в блоки 91812 и 91842 добавлена одна и та же Coinbase-транзакция d5d27987d2a3dfc724e359870c6644b40e497bdc0589a033220fe15429d88599 (рис. 4.29). Кроме того, в блоки 91722 и 91880 также добавлена одна и та же транзакция e3bf3d07d4b0375638d5f1db5255fe07ba2c4cb067cd81b84ee974b6585fb468 (рис. 4.30).

Транзакции блока ①

Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 133 bytes)	50.00000000 BTC
Хеш	d5d27987d2a3dfc724e359870c6644b40e497bdc0589a...	2010-11-14 20:59
COINBASE (Недавно Сгенерированные Монеты)	→	16va6NxJrMGe5d2LP6wUzuVnzBB... 50.00000000 BTC ⚡

Выходы ①

Индекс	0	Подробности Неизрасходованные
Адрес	16va6NxJrMGe5d2LP6wUzuVnzBB0KQZKom ⚡	Стоимость 50.00000000 BTC
Pkscript	046896ecfc449cb8560594eb7f413f199deb9b4e5d947a142e7dc7d2de0b811b8e204833ea2a2fd9d4c7b153a8ca7661d0a0b7fc981df1 f42f55d64b26b3da1e9c OP_CHECKSIG	

Рисунок 4.29. Дублирующая транзакция для блоков 91812 и 91842

Транзакции блока ①

Комиссия	0.00000000 BTC (0.000 sat/B - 0.000 sat/WU - 133 bytes)	50.00000000 BTC
Хеш	e3bf3d07d4b0375638d5f1db5255fe07ba2c4cb067cd81b...	2010-11-14 11:37
COINBASE (Недавно Сгенерированные Монеты)	→	1GktTvnY8KGfAS72DhzGYJRyaQNvYrK9Fg ... 50.00000000 BTC ⚡

Выходы ①

Индекс	0	Подробности Неизрасходованные
Адрес	1GktTvnY8KGfAS72DhzGYJRyaQNvYrK9Fg ⚡	Стоимость 50.00000000 BTC
Pkscript	04124b212f5416598a92cc88819105179dcb2550d571842601492718273fe0f2179a9695096bff94cd99dcccdea7cd9bd943bfca8fea649cac963411979a33e9 OP_CHECKSIG	

Рисунок 4.30. Дублирующая транзакция для блоков 91722 и 91880

Фактически произошло то, что майнер получил в качестве вознаграждения дважды одну и ту же транзакцию, то есть по факту вместо 100 BTC получил всего 50 BTC. Один из разработчиков системы Биткойн Рассел О'Коннор (Russell O'Connor) считает, что в случае с дублирующими транзакциями злоумышленник может удалять прошлые транзакции других пользователей из реестра [22].

Для того чтобы иметь возможность отслеживать дублирующие транзакции, в 2012 году были введены улучшения BIP30 [248] и BIP34 [55].

Важно отметить, что стандартные транзакции предназначены для защиты именно биткойн-сети, а не для того, чтобы защитить пользователя от совершения ошибок. Начиная с версии ядра системы Биткойн Bitcoin Core 0.9.3 стандартные транзакции также должны соответствовать следующим условиям:

1. Транзакция должна быть завершена: либо ее время блокировки должно быть в прошлом (или меньше, или равно текущей высоте блока), либо все ее порядковые номера должны иметь шестнадцатеричное значение FFFF FFFF.
2. Размер транзакции должен быть меньше 100 000 байт. Это примерно в 200 раз больше, чем типичная транзакция P2PKH с одним входом и одним выходом.
3. Каждый из скриптов подписи транзакции должен быть меньше 1650 байт. Этого достаточно, чтобы выполнить проверку 15 транзакций с несколькими подписями вида P2SH с использованием сжатых открытых ключей.
4. Открытые транзакции с несколькими подписями, которые не являются транзакциями типа P2SH и для которых требуется более трех открытых ключей, в настоящее время являются нестандартными.
5. Задача скрипта подписи транзакции заключается в том, чтобы поместить данные в стек для выполнения проверки. Он не может использовать новые коды операций, за исключением тех кодов операций, которые используются для помещения данных в стек.
6. Выход транзакции не может быть меньше, чем 1/3 от того количества сатоши, которое необходимо потратить в качестве комиссии за транзакцию. В настоящее время эта сумма составляет 546 сатоши для транзакций вида P2PKH или P2SH. Исключение составляют стандартные транзакции с нулевыми данными, у которых выход должен содержать нулевые сатоши.

Время блокировки и порядковый номер

Поле, в которое записано время блокировки транзакции (Locktime, в исходном коде ядра Bitcoin Core это поле называется nLockTime), может подписываться с любым типом хеш-кода, то есть подписывается всегда. Поле Locktime показывает самое раннее время, когда транзакция может быть добавлена в цепочку блоков.

Locktime позволяет подписывающим сторонам создавать транзакции с временной блокировкой, которые станут действительными только в будущем, давая подписавшимся сторонам возможность изменить свое мнение.

Если кто-либо из подписывающих сторон в транзакции с блокировкой (первой транзакции) передумает, он может создать новую (вторую) транзакцию без блокировки. Вторая транзакция будет использовать в качестве одного из своих входов один из входов первой транзакции. Это сделает первую транзакцию недействительной, если вторая транзакция будет добавлена в цепочку блоков до истечения срока блокировки.

Необходимо соблюдать осторожность перед истечением срока действия временной блокировки. Одноранговая сеть позволяет блокировать время за два часа до истечения реального времени, поэтому транзакция времени блокировки может быть добавлена в цепочку блоков за два часа до того, как ее временная блокировка официально истечет. Кроме того, блоки не создаются с гарантированными интервалами, поэтому любую попытку отменить ценную транзакцию следует предпринимать за несколько часов до истечения срока блокировки.

Предыдущие версии ядра Bitcoin Core использовали функцию, которая не позволяла подписывающим транзакциям использовать вышеописанный метод отмены транзакций. Но необходимая часть этой функции была отключена с целью предотвращения атаки типа «отказ в обслуживании».

Как следствие используемой ранее функции сейчас используются четырехбайтовые порядковые номера для обозначения каждого входа/выхода. Порядковые номера должны были позволить нескольким подписывающим сторонам прийти к соглашению об обновлении транзакции: когда они завершили обновление транзакции, они могли договориться установить порядковый номер каждого входа равным четырехбайтовому беззнаковому максимуму (шестнадцатеричное значение FFFF FFFF), что позволяло добавить транзакцию в блок, даже если ее временная блокировка еще не истекла.

Даже сегодня установка всех порядковых номеров в значение FFFF FFFF (как это происходит по умолчанию в Bitcoin Core) может отключить временную блокировку. Поэтому если вы хотите использовать время блокировки, то, по крайней мере, один вход должен иметь порядковый номер ниже максимального. Поскольку порядковые номера не используются сетью для каких-либо других целей, установки любого порядкового номера в ноль достаточно для включения времени блокировки.

Само время блокировки представляет собой 4-байтовое целое число без знака, которое можно проанализировать двумя способами:

- если значение числа меньше 500 миллионов, то время блокировки соответствует высоте блока; транзакцию можно добавить в любой блок, имеющий эту высоту или выше;
- если значение числа больше или равно 500 миллионам, то время блокировки рассчитывается с использованием формата времени эпохи Unix (количество секунд, прошедших со времени 0:00:00 1 января 1970 года; в настоящее время эта величина больше 1,395 миллиарда); транзакция может быть добавлена к любому блоку, время блока которого больше времени блокировки.

Отследить время блокировки можно с использованием различных сервисов. Например, сервис blockchair.com покажет вам все транзакции, которые в текущий момент находятся в пуле.

Можно настроить параметры отображения и вывести на экран интересующие поля транзакции, в том числе поле «Время блокировки». Также есть функция сортировки от большего к меньшему и наоборот.

Так, мы можем видеть, что на рис. 4.31 представлены транзакции или вообще без блокировки (в поле блокировки стоит значение 0), или указана высота блока (как правило, следующего блока по отношению к текущему); на рис. 4.31 представлены данные, актуальные на 15 января 2022 года.

Хеш	Входы (BTC)	Время (UTC)	Выходы (BTC)	Комиссия (USD)	Время блокировки
ab43a***33bc0	0.00293676	2022-01-15 21:53:00	0.00293472	0.09	0
64f82***90731	0.00148792	2022-01-15 21:53:00	0.00148462	0.14	0
5e65c***4d81b	0.01149833	2022-01-15 21:53:00	0.01148885	0.41	0
f16e3***b1db9	0.00073846	2022-01-15 21:53:00	0.00073012	0.36	718860
db6ff***cfc87	0.00103503	2022-01-15 21:53:00	0.00103059	0.19	0
a686d***37435	0.0105024	2022-01-15 21:53:00	0.01048775	0.63	0
867a6***8a8f2	0.40433213	2022-01-15 21:53:00	0.40312313	52.13	718860
942ad***1aadf	22,655.23142219	2022-01-15 21:52:59	22,655.23130219	5.17	0
941f6***9c6bf	0.00358305	2022-01-15 21:52:58	0.00357625	0.29	0
567a1***cd452	0.01181445	2022-01-15 21:52:57	0.01181002	0.19	718860

Рисунок 4.31. Транзакции по первому типу блокировки

Если же попробовать отсортировать транзакции по полю «Время блокировки» от большего к меньшему, то можно видеть на рис. 4.32, что самая первая транзакция имеет время блокировки, записанное в соответствии со временем Unix. Значение 1 642 282 318 указывает на время 16.01.2022 г. 00:31:58 в соответствии с часовым поясом Москва (+03:00). Выполнить подобный перевод можно с использованием любого онлайн-сервиса по конвертации времени, например [33].

Хеш	Время (UTC)	Выходы (BTC)	Комиссия (USD)	Время блокировки
658bc***952a2	2022-01-15 21:53:31	0.0025843	0.09	1642282318
00cba***3ecf6	2022-01-15 22:03:11	0.02195235	0.16	718861
f4780***3c37a	2022-01-15 22:01:38	0.08677175	0.44	718861
af05e***6da9d	2022-01-15 21:59:31	0.0933928	0.16	718861
df2af***7e674	2022-01-15 21:59:09	0.030993	0.17	718861
e36ba***0be35	2022-01-15 21:58:50	0.0271384	0.16	718861
bbace***0cc35	2022-01-15 21:57:07	0.07560367	0.60	718861
ef36d***bc1d2	2022-01-15 21:56:28	0.02208732	0.17	718861
fa605***b24b1	2022-01-15 21:55:00	0.07560367	0.40	718861
2ff39***5f1fe	2022-01-15 21:54:39	0.18534761	2.28	718861

Рисунок 4.32. Транзакции по второму типу блокировки

Комиссия за транзакцию

За то, что транзакция попадает в блок, может взиматься комиссия. Величина комиссии зависит от количества подписанных байтов транзакции.

Комиссия за каждый байт рассчитывается на основе текущего спроса на пространство в добытых блоках, при этом плата увеличивается по мере увеличения спроса. Комиссия за каждую транзакцию, попавшую в блок, начисляется тому майннеру, который этот блок создал. Каждый майннер устанавливает свое значение минимальной комиссии за каждую транзакцию, которую он будет принимать и пытаться замайнить.

Также существует концепция так называемых «высокоприоритетных транзакций». К высокоприоритетным транзакциям относятся транзакции, которые долгое время хранились в блокчейн-цепочке.

Считается, что длительный период хранения отражает величину приоритета транзакции. С таких транзакций комиссия не взимается. Однако каждый майннер сам решает, будет он работать с бесплатными транзакциями или нет. До версии ядра Bitcoin Core 0.12 для таких высокоприоритетных транзакций в каждом блоке было зарезервировано 50 КБ, однако теперь по умолчанию установлено значение 0 КБ.

Всем незамайненным транзакциям присваивается приоритет на основе суммы их комиссии за байт. При этом обработка начинается с более высокоплачиваемых транзакций. Транзакции добавляются в блок последовательно, пока не будет заполнено все доступное пространство.

Начиная с версии ядра Bitcoin Core 0.9 комиссия стала обязательным атрибутом транзакции. Для того чтобы транзакция передавалась по сети, требовалось уплатить минимальную комиссию (в настоящее время 1000 сатоши). Любая транзакция, оплачивающая только минимальную комиссию, может долгое время стоять в очереди ожидания, прежде чем в блоке будет достаточно свободного места для включения такой транзакции.

Величина комиссии автоматически вычитается из всех замайненных транзакций блока и добавляется к сумме вознаграждения майнера. На рис. 4.33 показана комиссия для блока 715 000, созданного 21 декабря 2021 года.

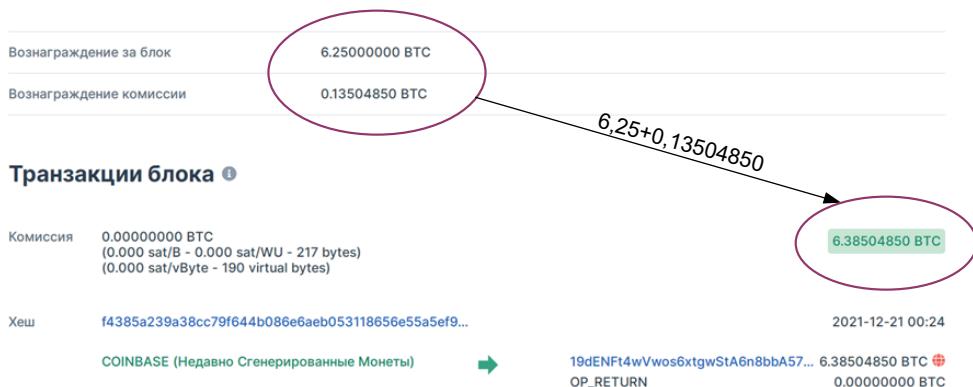


Рисунок 4.33. Комиссия для блока 715 000

Отследить сумму комиссии можно с использованием различных сервисов. Так же, как и со временем блокировки, сервис [blockchair.com](#) покажет вам все транзакции, которые в текущий момент находятся в пуле.

Можно настроить параметры отображения и вывести на экран интересующие поля транзакции, в том числе поле «Комиссия», а затем отсортировать транзакции по ее величине, например по убыванию (рис. 4.34).

Хеш	Время (UTC)	Комиссия (BTC)	Время блокировки
8706a***17f35	2022-01-15 22:24:15	0.00067	0
241b3***a2ff3	2022-01-15 22:24:36	0.00047634	0
56083***5ef3f	2022-01-15 22:24:35	0.00045	0
193ef***5efdc	2022-01-15 22:24:58	0.0004	0
27865***1af33	2022-01-15 22:24:52	0.00037179	0
d4058***e6512	2022-01-15 22:25:40	0.00035	0
b125e***2876f	2022-01-15 22:25:25	0.00022319	0
fce8b***8f6f8	2022-01-15 22:24:13	0.00018816	0
0132c***f7483	2022-01-15 22:25:08	0.00017851	0
cb2e6***93e6d	2022-01-15 22:24:59	0.000175	0

Рисунок 4.34. Транзакции с различной стоимостью комиссии в майнинг-пуле

Выход сдачи

Поскольку каждая транзакция тратит неизрасходованные выходы предыдущих транзакций (UTXO) и поскольку такие выходы можно потратить только один раз, то сумма всех неизрасходованных выходов, помещенная в транзакцию, должна быть или потрачена, или передана майнеру в качестве комиссии за транзакцию.

При этом очень редко получается так, что сумма неизрасходованных выходов точно соответствует той сумме, которую необходимо заплатить. Поэтому большинство транзакций содержат так называемый выход сдачи (change output).

Выход сдачи представляет собой обычный выход. Его особенность заключается в том, что в качестве адреса получателя данного выхода сдачи указывается пользователь, осуществляющий перевод. Таким образом ему в виде UTXO вернется обратно сдача от совершенной операции.

Пример транзакции со сдачей приведен на рис. 4.35, он относится к ранним транзакциям системы Биткойн. Транзакция 25252ecd1a0ffffbb294898d831cd5fe901ab7c4e6e181e3b9a889bf3640a6dc2 была создана 17 ноября 2014 года пользователем 122BNoyhmuUt9G9mdEm3mN4nb73c1UgNKt. По рис. 4.35 видно, что часть денег от суммы входов была переведена пользователю 16TBqpEFpivLtDRAtmNnCxEdJW1coodiia. Остальная сдача в количестве 110 BTC вернулась пользователю 122BNoyhmuUt9G9mdEm3mN4nb73c1UgNKt.

Резюме

USD BTC

Комиссия	0.00010000 BTC (26.738 sat/B - 6.684 sat/WU - 374 bytes)	199.99990000 BTC
Хеш	25252ecd1a0fffb294898d831cd5fe901ab7c4e6e181e3b9a889bf364... ↗	2014-10-17 05:55
122BNoyhmuUt9G9mdEm3mN4nb73c1UgNkT	49.99990000 BTC ↗	16TBqpEFpiVtDRAtmNhCxEdJW1coodlia
122BNoyhmuUt9G9mdEm3mN4nb73c1UgNkT	150.00001000 BTC ↗	89.99990000 BTC ↗ 122BNoyhmuUt9G9mdEm3mN4nb73c1UgNkT 110.00000000 BTC ↗

Рисунок 4.35. Пример транзакции с выходом сдачи**Как избежать повторного использования ключей**

В транзакции и отправитель, и получатель раскрывают друг другу все открытые ключи или адреса, применяемые в транзакции. Это позволяет любому человеку использовать общедоступную цепочку блоков для отслеживания прошлых и будущих транзакций с применением тех же открытых ключей или адресов какого-либо пользователя.

Если один и тот же открытый ключ часто используется повторно, как это происходит, когда люди используют биткойн-адреса – 20-байтовые дважды хешированные открытые ключи по формуле RIPEMD-160(SHA-256(*public_key*)) – в качестве статических платежных адресов, другие люди могут легко отслеживать историю поступлений и трат такого пользователя, включая знание о том, сколько денег доступно в настоящий момент адресу данного пользователя. Несмотря на то что блокчейн является открытой системой, такая открытость ставит под угрозу пользователей сети Биткойн, делает их уязвимыми.

Поэтому считается, что правильно – использовать каждый открытый ключ ровно дважды: один раз для получения платежа и один раз для его расходования. В таком случае пользователь может получить значительную финансовую конфиденциальность.

Более того, использование новых открытых ключей или уникальных адресов при приеме платежей или создании выходов сдачи можно комбинировать с другими методами, для того чтобы еще сильнее запутать следы и предотвратить возможность использования цепочки блоков для отслеживания того, как пользователи получают и тратят свои деньги.

Предотвращение повторного использования ключей также может обеспечить дополнительную защиту от потенциальных атак по реконструкции закрытых ключей из открытых ключей (однако вероятность практической реализации подобных атак для используемого алгоритма ECDSA крайне мала) или по сравнению подписей (возможно сегодня при определенных обстоятельствах, описанных далее, с предположительным использованием в контексте более общих атак).

Уникальные (не используемые повторно) адреса для транзакций типа P2PKH и P2SH защищают от первого типа атак, сохраняя открытые ключи ECDSA скрытыми (хешированными) до тех пор, пока не будут израсходованы монеты, отправленные на эти адреса в первый раз, поэтому атаки фактически бесполезны, если они не могут восстановить закрытые ключи в течение одного-двух часов, пока транзакция не будет надежно встроена в цепочку блоков.

Уникальные (не используемые повторно) закрытые ключи защищают от второго типа атак путем создания только одной подписи для каждого закрытого ключа, в результате чего злоумышленники никогда не получают последую-

щую подпись для использования в атаках на основе сравнения. Существующие атаки, основанные на сравнении, сегодня практически только тогда, когда при подписании используется недостаточная энтропия для генерации случайных чисел или когда используемая энтропия раскрывается некоторыми средствами, такими как атаки по побочным каналам. Пример описания подобной атаки был приведен ранее в разделе 2.3.3.

Подводя итог, можно сказать, что как для обеспечения конфиденциальности транзакций, так и для их безопасности в общем разработчики системы Биткойн рекомендуют избегать повторного использования открытого ключа и адресов в сети Биткойн.

Гибкость транзакции

Ни один из типов подписываемого хеш-кода в системе Биткойн не защищает скрипты подписи, оставляя возможность злоумышленнику применить атаку типа «отказ в обслуживании». Данное свойство называется «гибкостью транзакции» (Transaction Malleability).

Скрипт подписи содержит подпись в соответствии с алгоритмом ECDSA, которая не может подписывать сама себя. Это позволяет злоумышленникам вносить нефункциональные изменения в транзакцию (в те поля, которые не подлежат подписи), при этом сама транзакция остается действительной.

Например, злоумышленник может добавить некоторые данные в скрипт подписи, которые будут удалены до обработки предыдущего скрипта открытого ключа.

Хотя модификации нефункциональны (то есть они не меняют входы и выходы, которые используются в транзакции), они тем не менее влияют на изменение хеш-кода транзакции. Так как идентификатор транзакции (txid) является значением хеш-кода этой транзакции, то ее создатель будет пытаться использовать исходный идентификатор для дальнейшего управления выходами. Однако если злоумышленник внесет изменения в транзакцию и, как следствие, изменит идентификатор транзакции, создатель транзакции этого сделать уже не сможет.

Подобная ситуация не является проблемой для большинства биткойн-транзакций, которые предназначены для немедленного добавления в цепочку блоков. Однако это становится проблемой, когда выходные данные транзакции изменяются до того, как транзакция добавляется в цепочку блоков. Разработчики сети Биткойн постоянно работают над снижением гибкости транзакций среди стандартных типов транзакций.

Так, при использовании улучшения BIP62 предлагаются изменения в правилах проверки действительности транзакций системы Биткойн, чтобы сделать гибкость транзакций невозможной [249].

Еще одним вариантом улучшения является BIP141: Segregated Witness [168], которое поддерживается ядром Bitcoin Core и было активировано в августе 2017 года. Улучшение BIP141 определяет новую структуру, называемую «свидетельство» (witness), которая фиксируется в блоках отдельно от транзакций дерева Меркля. Эта структура содержит данные, необходимые для проверки действительности транзакции. В частности, в эту новую структуру перемещены скрипты и подписи.

Получается, что теперь не входы и выходы содержат данные для проверки, а специально отведенное для этого поле witness. Помимо того что теперь злого-

умышленник не может внести изменения в сами инструкции, сократился и общий объем, занимаемый одной транзакцией. Таким образом, в настоящий момент блок биткойна может вмещать около 2700 вместо 1650 транзакций [52]. Это значительно повысило пропускную способность сети Биткойн.

С введением Segregated Witness поменялась форма скрипта открытого ключа для транзакций типа транзакций P2PKH и P2SH (важно отметить, что поменялась форма записи, но не смысл самих проверок).

Так, вместо P2PKH теперь используется форма транзакции P2WPKH (Pay-To-Witness-Public-Key-Hash – «Плата за свидетельство хеша открытого ключа»). Для транзакции P2WPKH скрипт открытого ключа содержит значение 0, за которым следует 20-байтовый хеш-код ключа, скрипт подписи остается пустым, а поле witness содержит подпись и открытый ключ [37]. Основные поля транзакции P2WPKH приведены в табл. 4.10.

Таблица 4.10. Основные поля транзакции P2WPKH

Скрипт открытого ключа	OP_0 <keyhash (20 byte)>
Скрипт подписи	Пустое поле
Witness	<sig><pubkey>

На рис. 4.36 представлен пример транзакции типа P2WPKH, где выход № 0 транзакции 1d8149eb8d8475b98113b5011cf70e0b7a4dccff71286d28b8b4b641f94f1e46 превращается во вход № 24 транзакции 5944d4ff0e5620c7933c9a3c15c21f2dd33a6f8192a54109711a8a374a6bf1e3.

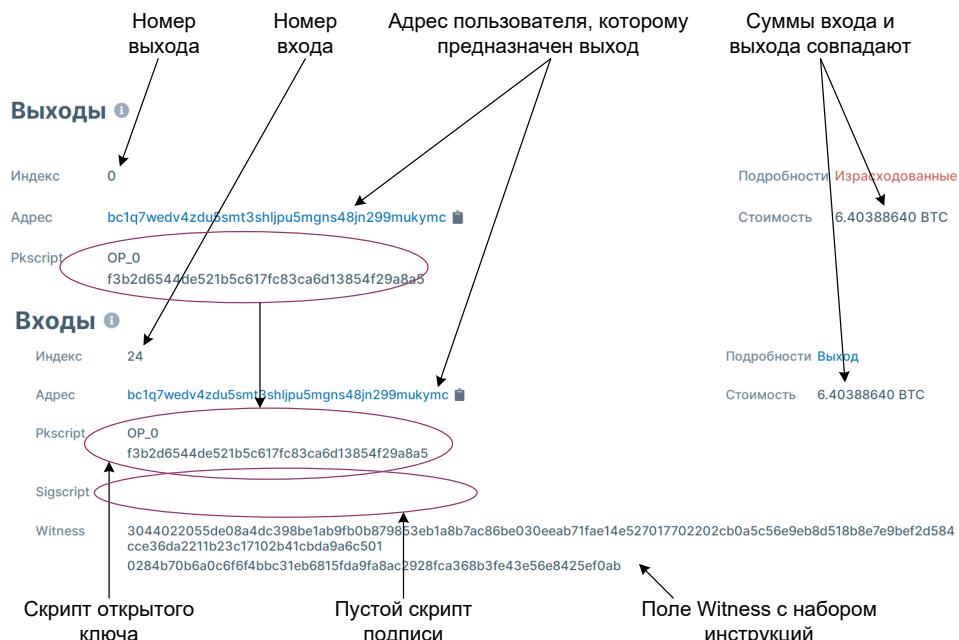


Рисунок 4.36. Пример перехода выхода во вход для транзакции типа P2WPKH

Аналогичным образом в качестве транзакции P2SH теперь используется форма транзакции P2WSH (Pay-To-Witness-Script-Hash – «Плата за свидетельство хеша скрипта»). Для транзакции P2WSH скрипт открытого ключа содержит значение 0, за которым следует 32-байтовый хеш скрипта witness, скрипт подписи остается пустым, а поле witness содержит в качестве префикса нулевое значение, за которым следуют подпись, открытые ключи и инструкция для проверки (могут быть варианты как для одной подписи, так и для мультиподписи) [37].

Поле witness для транзакции P2WSH делится на две части. Первая часть содержит список подписей, с помощью которых доказывается владение монетами. Во второй части размещается скрипт witness (Witness Script), содержимое которого определяет правила траты.

Основное отличие заключается в том, что содержимое скрипта теперь указывается в момент траты монет (при создании входа), а в момент отправки монет (при создании выхода) указывался только хеш-код от скрипта witness. Основные поля транзакции P2WSH приведены в табл. 4.11.

Таблица 4.11. Основные поля транзакции P2WSH

Скрипт открытого ключа	OP_0 <WitnessScriptHash (32 byte)>
Скрипт подписи	Пустое поле
Witness	0 <sig1>< 1 <pubkey1><pubkey1><pubkey2> 2 OP_CHECKMULTISIG>

На рис. 4.37 представлен пример транзакции типа P2WSH, где выход № 0 транзакции af5c739ff9b1ff8921bb7aced0d473570d6ed7bed0171b4f216bd526951c0cf0 превращается во вход № 0 транзакции 55e16683f14b4ae4b4652c8b459127cb8823c06ed030c1349fded95ab1bbbf0b.

Так как новый формат отличается от старого, то для обратной совместимости (чтобы старые сервисы и кошельки могли обрабатывать новые транзакции) используется специальная структура, которая позволяет сформировать транзакцию, обладающую свойствами Segregated Witness с одной стороны, но не отличается от структуры P2SH с другой стороны [37]. Основные поля транзакции P2WSH с обратной совместимостью приведены в табл. 4.12.

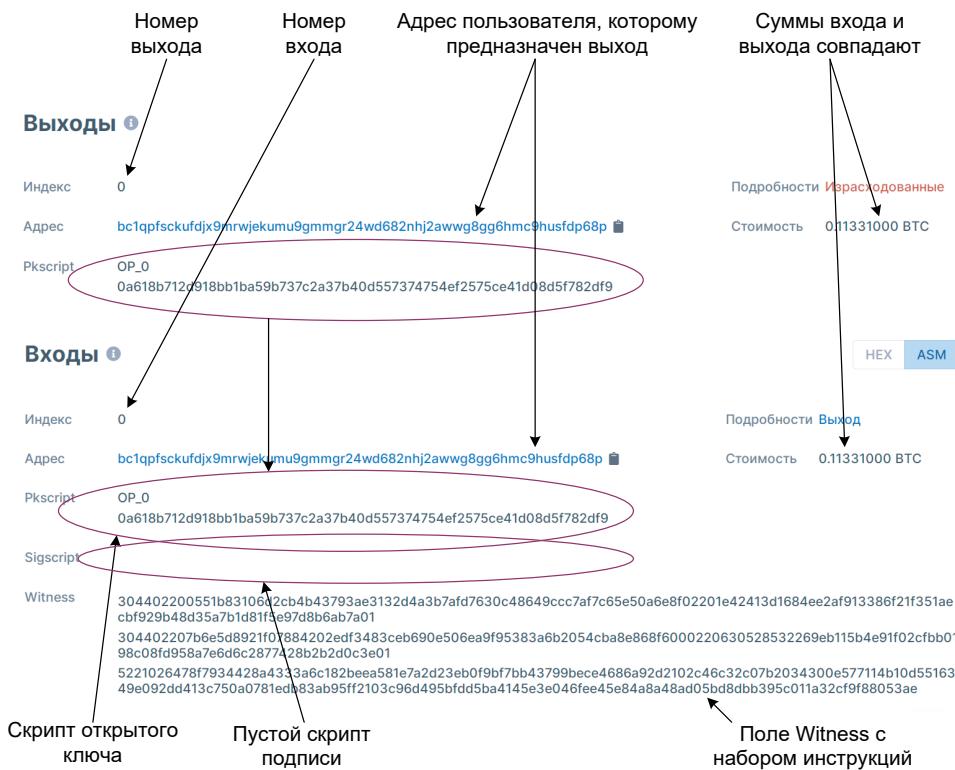


Рисунок 4.37. Пример перехода выхода во вход для транзакции типа P2WSH

Таблица 4.12. Транзакция P2WSH с обратной совместимостью

Скрипт открытого ключа	OP_HASH160 <ScriptHash (20 byte)> OP_EQUAL
Скрипт подписи	OP_0 <keyHash (20 byte)>
Witness	<sig><pubkey>

На рис. 4.38 представлен пример транзакции типа P2WSH с обратной совместимостью, где выход № 0 транзакции превращается во вход № 0 другой транзакции.

Гибкость транзакций также влияет на отслеживание платежей. Интерфейс системы Биткойн позволяет отслеживать транзакции по их идентификатору, но если этот идентификатор изменится из-за изменения транзакции, то может показаться, что транзакция исчезла из сети.

Опыт работы с системой Биткойн по отслеживанию транзакций показывает, что самым предпочтительным способом отслеживания транзакции является отслеживание с помощью выходов UTXO, так как их нельзя изменить без признания транзакции недействительной. Также необходимо учитывать, что если транзакция действительно исчезает из сети и ее необходимо перевыпустить, то ее следует повторно выпустить таким образом, чтобы сделать потерянную транзакцию недействительной. Самый действенный метод в данном

случае – убедиться, что новая транзакция использует все те же выходы, которые потерянная транзакция использовала в качестве входов.

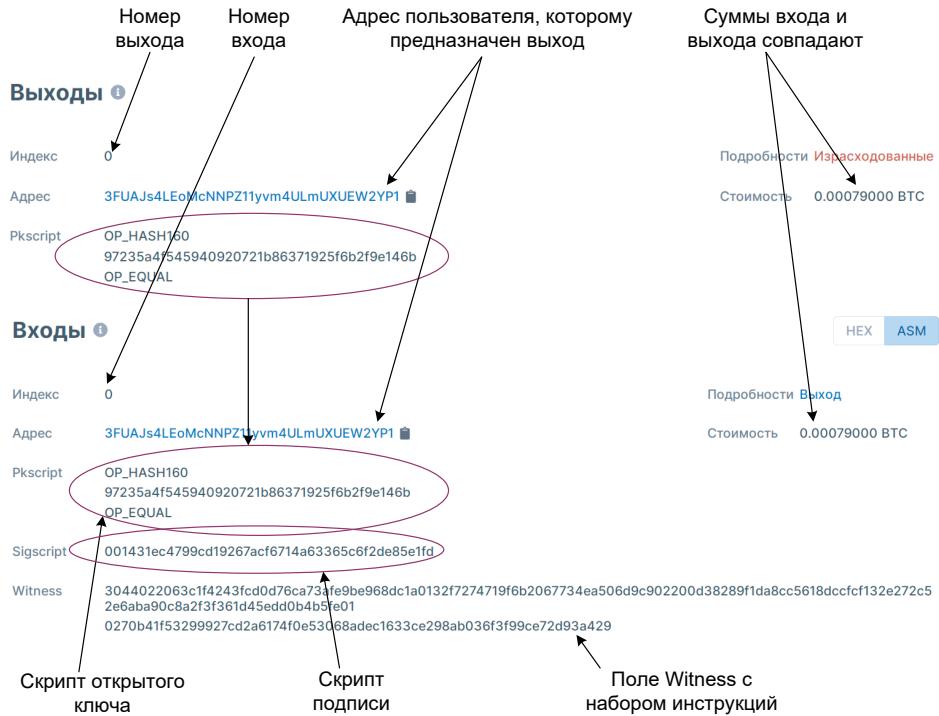


Рисунок 4.38. Пример перехода выхода во вход для транзакции типа P2WSH с обратной совместимостью

Смарт-контракты в системе Биткойн

Смарт-контракт представляет собой транзакцию особого типа. Такие транзакции в системе Биткойн используются для обеспечения соблюдения финансовых соглашений. Предназначение контрактов в первую очередь заключается в том, чтобы избежать участия посредников при совершении сделок.

Контракт условного депонирования и арбитраж

Рассмотрим пример контракта между некоторыми продавцом и покупателем [99]. У Алисы есть товар, и она хочет продать его Бобу. У Боба есть деньги, и он хочет купить товар Алисы. Но оба они не доверяют друг другу. В этом случае они могут использовать контракт, чтобы гарантировать, что Боб получит свой товар, а Алиса получит оплату.

Самый простой вариант контракта: Алиса сможет потратить выход Боба с указанной суммой только в том случае, если Алиса и Боб оба подпишут входную транзакцию. Это означает, что Алисе не заплатят, если Боб не получит свой товар, и Боб не сможет оставить товар себе и получить при этом деньги Алисы.

Однако этот простой контракт будет работать в том случае, если Алиса и Боб являются честными партнерами и у них не возникает споров. Поэтому для ре-

шения споров в контракт добавляется третий игрок – арбитр Кэрол, а также создается контракт условного депонирования (эскроу).

Боб из своих денег создает выход, который можно потратить, только если двое из трех человек подпишут вход. Теперь Боб может заплатить Алисе, если все в порядке; Алиса может вернуть деньги Бобу, если возникнет проблема, или Кэрол может выступить в качестве арбитра и решить, кто должен получить деньги, если возникнет спор.

Чтобы создать выход с несколькими подписями (multisig), все участники контракта должны обменяться своими открытыми ключами. После этого покупатель (Боб) создает следующий скрипт погашения мультиподписи P2SH (в скрипте не показаны коды операций для помещения открытых ключей в стек):

```
OP_2 [Открытый ключ Алисы] [Открытый ключ Боба] [Открытый ключ Кэрол] OP_3
OP_CHECKMULTISIG
```

Коды операций OP_2 и OP_3 помещают в стек числа 2 и 3 соответственно. OP_2 указывает, что для исполнения транзакции достаточно наличия двух подписей; OP_3 указывает, что предоставляются три открытых ключа (без хеширования).

Данный скрипт представляет собой скрипт открытого ключа с несколькими подписями «2 из 3», в более общем смысле называемый скриптом открытого ключа m из n (где m – минимальное количество требуемых совпадающих подписей, а n – количество предоставленных открытых ключей).

Алиса передает скрипт погашения Бобу, который проверяет, что в скрипт включены все открытые ключи, включая открытый ключ арбитра Кэрол. Боб хеширует скрипт погашения для создания скрипта погашения транзакции P2SH и переводит деньги на счет этого скрипта погашения. Алиса видит, что платеж добавляется в цепочку блоков, и после этого отправляет товар.

Допустим, при транспортировке товара с ним неосторожно обращались и часть товара была повреждена. Боб требует полного возмещения денег, но Алиса считает, что достаточно компенсировать 20 % от стоимости товара. Они обращаются к Кэрол, чтобы решить проблему. Кэрол просит у Боба фото-свидетельство вместе с копией скрипта погашения, созданного Алисой и проверенного Бобом. Изучив доказательства, Кэрол считает, что возврата 30 % достаточно, поэтому она создает и подписывает транзакцию с двумя выходами, один из которых переводит 70 % денег на открытый ключ Алисы, а второй переводит оставшиеся 30 % на открытый ключ Боба.

В скрипт подписи Кэрол помещает свою подпись и копию нехешированного скрипта погашения, созданного Алисой. Она передает копию незавершенной транзакции Алисе и Бобу. Любой из них может завершить ее, добавив свою подпись для создания следующего скрипта подписи (в скрипте не показаны коды операций для отправки подписей и скрипта погашения в стек):

```
OP_0 [подпись Алисы] [подпись Боба или Кэрол] [Скрипт погашения]
```

OP_0 – это обходной путь для устранения единичной ошибки в исходной реализации, которую необходимо сохранить для совместимости. Обратите внимание, что скрипт подписи должен содержать подписи в таком же порядке, в котором они были расположены в скрипте погашения.

Когда транзакция передается по сети, каждый узел сверяет скрипт подписи с выходом транзакции P2SH, ранее сформированным Бобом, гарантируя, что скрипт погашения действительно соответствует ранее предоставленному хеш-коду скрипта погашения.

После этого выполняется проверка скрипта погашения, при этом в качестве входных данных используются две подписи. Если скрипт погашения проходит проверку, то два выхода транзакции отображаются в кошельках Алисы и Боба как неизрасходованные остатки (UTXO).

В случае если Кэрол создала и подписала транзакцию, на которую ни Алиса, ни Боб не согласны (например, Кэрол создала выход, который переводит все деньги на ее счет), то Алиса и Боб могут найти нового арбитра и создать новую транзакцию с новым скриптом погашения с несколькими подписями в формате 2 из 3. Это позволяет Алисе и Бобу не беспокоиться о том, что арбитр может украсть их деньги.

Для удобства работы с такими типами контрактов используются специальные приложения. Например, ресурс BitRated [82] предоставляет интерфейс службы арбитража с несколькими подписями, разработанный с использованием средств HTML/JavaScript на веб-сайте с лицензией GNU AGPL.

Канал микроплатежей

Рассмотрим другой сценарий [99]. Пусть Алиса делает для Боба сдельную работу. Например, она следит за работой сайта Боба: размещает новости, обновляет контент, отвечает на обращения пользователей. Боб обещал платить Алисе за каждое действие, совершенное ею на форуме.

Однако по факту оказалось, что Боб часто забывает заплатить ей. Поэтому Алиса требует оплаты сразу после каждого выполненного действия. Боб отказывается пересыпал транзакции за каждое выполненное действие, так как в этом случае ему необходимо будет платить комиссию за каждую транзакцию. В этом случае на помощь Алисе и Бобу приходит канал микроплатежей.

Боб запрашивает у Алисы ее открытый ключ, а затем создает две транзакции. Первая транзакция создает перевод 100 миллибиткойнов на выход транзакции P2SH, для которой скрипт погашения с мультиподписью 2 из 2 требует подписей как Алисы, так и Боба.

Данная транзакция является транзакцией с долговым обязательством. Трансляция этой транзакции в сеть позволит Алисе удерживать миллибиткойны Боба в заложниках в течение времени блокировки (в случае данного примера это 24 часа).

Боб пока сохраняет эту транзакцию у себя и создает вторую транзакцию. Вторая транзакция возвращает все миллибиткойны первой транзакции (за вычетом комиссии за транзакцию; на рис. 4.39 показана схема без учета комиссии) обратно Бобу после 24-часовой задержки, установленной временем блокировки.

Данная транзакция является транзакцией возврата. Боб не может подписать транзакцию возврата самостоятельно, поэтому он передает ее Алисе на подпись, как показано на рис. 4.39 (перевод рисунка с сайта [99]).

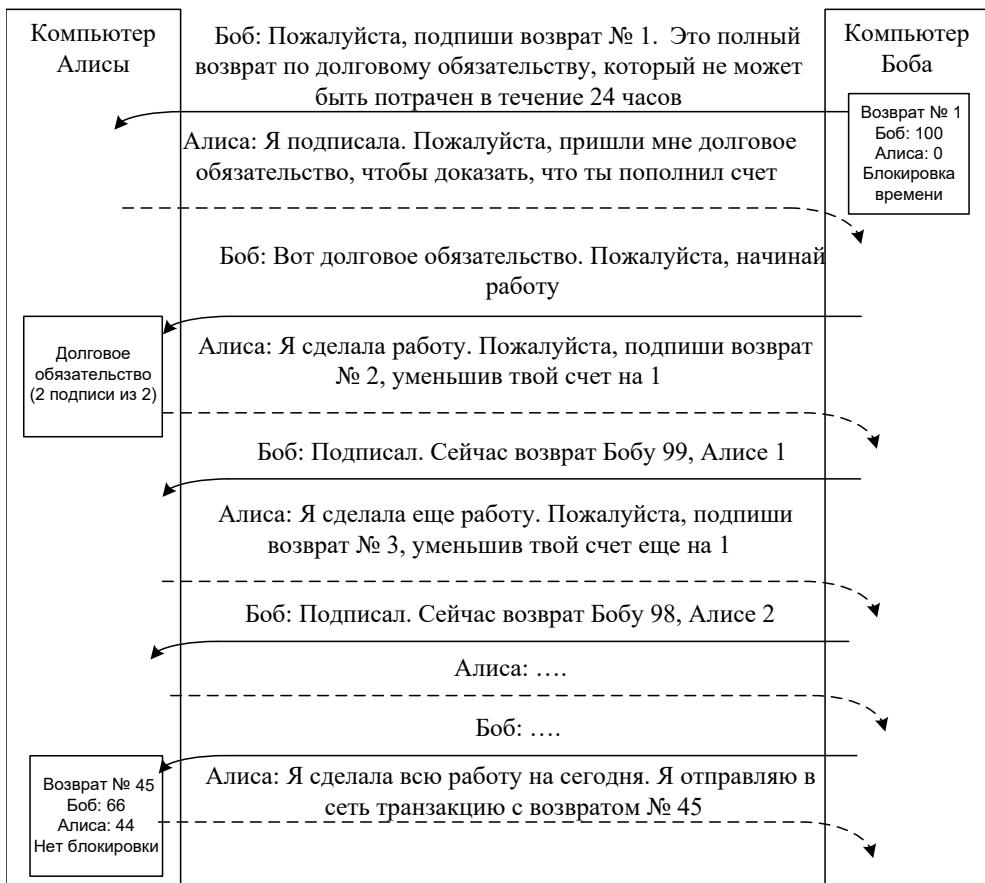


Рисунок 4.39. Схема взаимодействия Алисы и Боба при формировании канала микроплатежей

Алиса проверяет, что первая транзакция Боба имеет блокировку на 24 часа (а значит, Боб не сможет потратить эту транзакцию в ближайшие сутки), подписывает ее и возвращает копию Бобу. Затем она спрашивает Боба о транзакции с долговым обязательством.

Боб формирует вторую транзакцию, в которой он в качестве входных данных указывает выход первой транзакции, по-прежнему переводя на свой счет полную сумму из транзакции номер 1. Алиса следит за тем, чтобы вход второй транзакции совпадал с выходом первой транзакции.

Если Алиса сейчас подпишет эту транзакцию в сеть, то Бобу придется выждать 24 часа, прежде чем он сможет потратить свои деньги. Если Алиса ничего не сделает, Боб также сможет тратить свои деньги, после того как истекут 24 часа. По факту в этот момент Боб еще ничего не теряет (кроме, возможно, небольшой комиссии за транзакцию).

Теперь каждый раз после того, как Алиса выполняет какую-то работу стоимостью 1 миллибиткойн, она просит Боба создать и подписать новую версию

транзакции долгового обязательства. Вторая версия транзакции переводит 1 миллибиткойн Алисе, а остальные 99 – обратно Бобу.

У транзакции долгового обязательства нет времени блокировки, поэтому Алиса может сразу подписать эту транзакцию и отправить в сеть. Но она этого не делает, так как сделала еще не всю запланированную работу. Алиса и Боб повторяют шаги работы и оплаты до тех пор, пока Алиса не закончит рабочий день или пока не истечет срок действия временной блокировки.

После этого Алиса подписывает окончательную версию транзакции долгового обязательства и отправляет ее в сеть. Таким образом Алиса получает ту сумму, которую она успела заработать, а Боб получает остаток. На следующий день, перед тем как Алиса приступит к работе, они создадут новый канал микроплатежей.

Если Алиса не успеет отправить в сеть блокчейна (включая процесс майнинга) свою транзакцию с переводом заработанных денег на свой счет до того, как первая транзакция Боба будет разблокирована, то Боб может успеть получить полный возврат своих денег. Это одна из причин, по которой каналы микроплатежей лучше всего подходят для небольших платежей. Если, например, у Алисы возникнут проблемы с интернетом ближе к истечению срока действия временной блокировки, она не сможет отправить свою транзакцию в сеть и, как следствие, ее платеж может оказаться недействительным.

Считается, что для больших платежей сумма комиссии ничтожно мала (по сравнению с суммой перевода) и поэтому имеет смысл сразу отправлять транзакцию с переводом денег в сеть Биткойн.

На сайте [bitcoinj](#) [83] представлена библиотека для языка программирования Java, которая содержит полный набор функций микроплатежей. Пример реализации микроплатежей с помощью данной библиотеки приведен в специальном руководстве [247].

Транзакции объединения монет

Транзакции объединения монет (CoinJoin) используются тогда, когда пользователи системы Биткойн не хотят оставлять прямой след своих трат и поступлений. Так как сеть Биткойн является публичной сетью, то любой пользователь может отслеживать перемещение средств между адресами ее участников. Для того чтобы усилить конфиденциальность пользователей и затруднить отслеживание движения денежных средств, была придумана транзакция вида CoinJoin.

Транзакция объединения монет создается несколькими пользователями, которые предварительно об этом договорились. Все пользователи соглашения передают на вход транзакции одинаковое количество монет (рис. 4.40). На выходе транзакции CoinJoin формируется несколько выходов с одинаковой суммой (по количеству участников соглашения). При этом выходы назначаются на новые адреса системы Биткойн, ранее сгенерированные участниками соглашения.

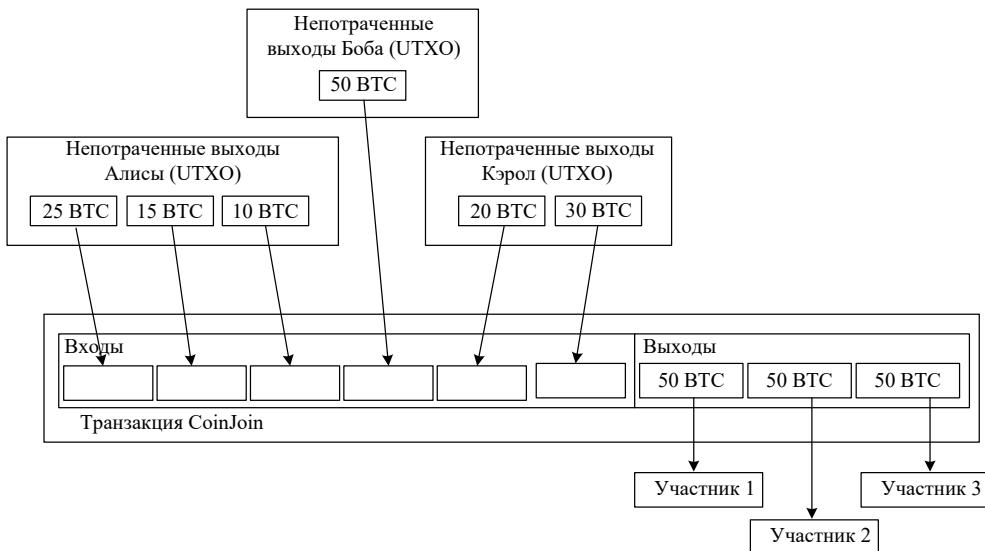


Рисунок 4.40. Пример транзакции объединения монет

Рассмотрим более подробно транзакцию, пример которой приведен на рис. 4.40. Алиса, Боб и Кэрол решают создать транзакцию CoinJoin. Каждый из них подготавливает входы на общую сумму 50 BTC. Кроме того, каждый из них готовит новые адреса в сети Биткойн (на рис. 4.40 это Участник 1, Участник 2 и Участник 3).

Эти адреса и данные входов передаются одному из участников соглашения – фасилитатору. Допустим, фасилитатором является Алиса. Тогда Боб и Кэрол должны передать Алисе данные своих входов и хеш-кодов открытых ключей.

Алиса создает транзакцию типа CoinJoin, расходующую каждый из UTXO на три выхода одинакового размера. Затем Алиса подписывает транзакцию с использованием типа подписываемого хеш-кода *SIGHASH_ALL*, для того чтобы никто не мог изменить входные или выходные данные.

Она передает частично подписанную транзакцию Бобу, который подписывает свои данные таким же образом. После этого Боб передает транзакцию Кэрол, которая также подписывает ее. После этого Кэрол отправляет транзакцию в сеть Биткойн, для того чтобы она попала в блок.

Таким образом, получается, что никто, кроме Алисы, Боба и Кэрол, не знает новые адреса друг друга и потому не может проследить дальнейшее расходование средств. Если выполнить несколько транзакций CoinJoin, то след будет запутан еще сильнее, что повысит конфиденциальность для средств участников таких транзакций.

В рассмотренном выше варианте транзакции CoinJoin участники должны будут выплатить некоторое вознаграждение за транзакции. Альтернативой этому служит метод, когда несколько пользователей хотят совершить покупку или перевод на одинаковую сумму денег. Тогда они могут для выходных транзакций вместо новых адресов участников сети использовать непосредственно адреса своих продавцов.

Чем больше пользователей принимает участие в сделке с транзакцией CoinJoin, тем сложнее проследить историю переводов по сети Биткойн, но тем больше пользователей будут посвящены в детали сделки.

Считается [4], что самой масштабной анонимной операцией с использованием транзакций CoinJoin была транзакция на сумму 34,30693136 BTC (около 111,5 млн рублей на январь 2022 года). Операция была приурочена к юбилейному событию – 10-летию со дня публикации White Paper биткойна [187]. Эта транзакция была создана 1 ноября 2018 года в 1:43 и имеет хеш-код 940ee6db84456b34a419112f71394f81c236873fb5262ae2398e29a1171475f. Подробности приведены на рис. 4.41.

Комиссия	0.00018675 BTC (1.288 sat/B - 0.569 sat/WU - 12943 bytes) (2.276 sat/vByte - 7326 virtual bytes)	34.30693136 BTC
Хеш	940ee6db84456b34a419112f71394f81c236873fb5262ae2398e29a1171475f	2018-11-01 01:43

bc1q750eus2apv4ex0aurj6q532tqyuz4f54d5sf	0.0003401 BTC	0.10002606 BTC
bc1qap4s0w72hc2fe70w07k6uyrpj0l4y9n3kjzdmn7	0.05003244 BTC	0.10002606 BTC
bc1qjyxhmsgyy78lmz54eqry7237ylcwemesthqj3	0.48408319 BTC	0.10002606 BTC
bc1qy3vtcpmwkxsh2w0eyhdwcx7d2fqwdluzafu	0.09259491 BTC	0.10002606 BTC
bc1qnm5p276e9sd0n097uddgechds4pfmr9urn...	0.00741509 BTC	0.10002606 BTC
bc1q6u6qqfj78fkpkvuw440ffxy3ecpetc5n2jhsg6v	0.00030003 BTC	0.10002606 BTC
bc1qgcmrkvp97ryyfuj3ucrp6883vcuinqv2960nc	14.82915536 BTC	0.10002606 BTC
bc1q66hnj5hnum7upl8yym8up8rfqyse65j02e7vh	0.10381237 BTC	0.10002606 BTC
bc1qt2c5r7rgammgt7pedgncc95xhcuwiy2ptz2s	0.50000000 BTC	0.10002606 BTC
bc1qt585gphexiz5quuyi583z34qx5rljwvv3e5g	0.34981402 BTC	0.10002606 BTC

Загрузить больше входов... (осталось 60)

Загрузить больше выходов... (осталось 73)

Рисунок 4.41. Большая анонимная транзакция CoinJoin

4.1.5 Кошельки в системе Биткойн

Помимо описанных в главе 3 видов кошельков, для системы Биткойн характерно использование кошельков-файлов, которые применяются для хранения в цифровом виде в файле набора закрытых ключей.

Рассмотрим формат закрытого ключа. В системе Биткойн закрытый ключ в стандартном формате – это 256-битовое число между значениями 1 и FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140 (в шестнадцатеричном виде), что представляет собой почти весь диапазон значений $2^{256} - 1$. Данный диапазон регулируется стандартом электронной подписи ECDSA на основе эллиптической кривой secp256k1, который был рассмотрен в главе 2.

Формат импорта кошелька

Формат импорта кошелька (Wallet Import Format, WIF) используется для того, чтобы сделать копирование закрытых ключей менее подверженным ошибкам. Данный формат использует кодировку Base58Check [61] для закрытого ключа, что значительно снижает вероятность ошибки копирования.

Преобразование закрытого ключа в формат импорта кошелька WIF может быть выполнено следующим образом:

1. Возьмите закрытый ключ.
2. Добавьте перед ним байт со значением 80 (значения байтов приведены в шестнадцатеричном виде) для адресов основной сети или со значением EF для адресов тестовой сети.

3. Добавьте после него единичный байт (байт со значением 01), если закрытый ключ должен использоваться со сжатыми открытыми ключами. Ничего не добавляется, если он используется с несжатыми открытыми ключами.
4. Выполните хеширование в соответствии с алгоритмом SHA-256 (см. главу 1) для расширенного ключа.
5. Выполните хеширование по алгоритму SHA-256 для результата предыдущего хеширования.
6. Возьмите первые четыре байта второго хеш-кода SHA-256; они являются контрольной суммой.
7. Добавьте четыре байта контрольной суммы в конец расширенного (в соответствии с пп. 1 и 2) ключа.
8. Преобразуйте результат из байтовой строки в строку Base58, используя кодировку Base58Check.

Этот процесс легко обратим с использованием функции декодирования Base58.

Существует также формат закрытого мини-ключа [181], который представляет собой метод кодирования закрытого ключа длиной менее 30 символов, позволяющий встраивать ключи в небольшое физическое пространство, такое как физические токены биткойнов и более устойчивые к повреждениям QR-коды.

Формат BIP39

BIP39 описывает шаги, которые необходимо предпринять, чтобы превратить приватный ключ в мнемоническую фразу. Это предложение стало стандартом для кошельков [194].

Последовательность действий состоит из двух частей: генерации мнемоники и ее преобразования в бинарное начальное число. Позже это начальное число можно использовать для создания детерминированных кошельков с использованием различных существующих методов.

Мнемонический код или предложение лучше подходят для взаимодействия с человеком по сравнению с необработанными двоичными или шестнадцатеричными представлениями исходного числа кошелька. Человеческий мозг легче запоминает осознанные слова, которые можно с чем-то ассоциировать, нежели случайный набор букв и цифр. Мнемоническую фразу можно записать на бумаге, легко запомнить или передать по телефону.

Рассмотрим основные принципы, которые лежат в основе построения мнемонической фразы (в соответствии с BIP39) и дальнейшей выработки ключа с ее использованием.

Мнемоническая фраза может состоять из 12 или 24 слов. В зависимости от этого используются различные параметры выработки. Мы далее будем записывать параметры в формате x/y , где x – параметр для мнемонической фразы из 12 слов, а y – параметр для мнемонической фразы из 24 слов.

На текущий момент (середина 2022 г.) существует возможность задать мнемоническую фразу на одном из 10 языков: это английский, японский, корейский, испанский, китайский упрощенный, китайский традиционный, французский, итальянский, чешский и португальский языки [76].

В каждом языковом словаре представлено 2048 слов. Несмотря на то что строки в словаре пронумерованы от 1 до 2048, необходимо помнить о том, что при двоичном представлении данных нумерация будет вестись от 0 до 2047. Так, для словаря английского языка 11-битовая последовательность из всех нулей (00000000000) будет соответствовать первому слову словаря – abandon, а 11-битная последовательность из всех единиц (11111111111) будет соответствовать последнему слову zoo.

Для того чтобы сгенерировать мнемоническую фразу из 12/24 слов, используется следующий алгоритм [193] (рис. 4.42):

1. Сгенерировать 128/256-битовую случайную последовательность M .
2. Получить хеш-код $\text{Hash}(M)$.
3. Составить 132/264-битовую последовательность, соединив 128/256 бит последовательности M и первые 4/8 бит от хеш-кода $\text{Hash}(M)$.
4. Разделить 132/264-битовую последовательность на 12/24 последовательностей по 11 бит.
5. Заменить каждую 11-битовую последовательность в соответствии со словарем на соответствующее слово и получить мнемоническую фразу.

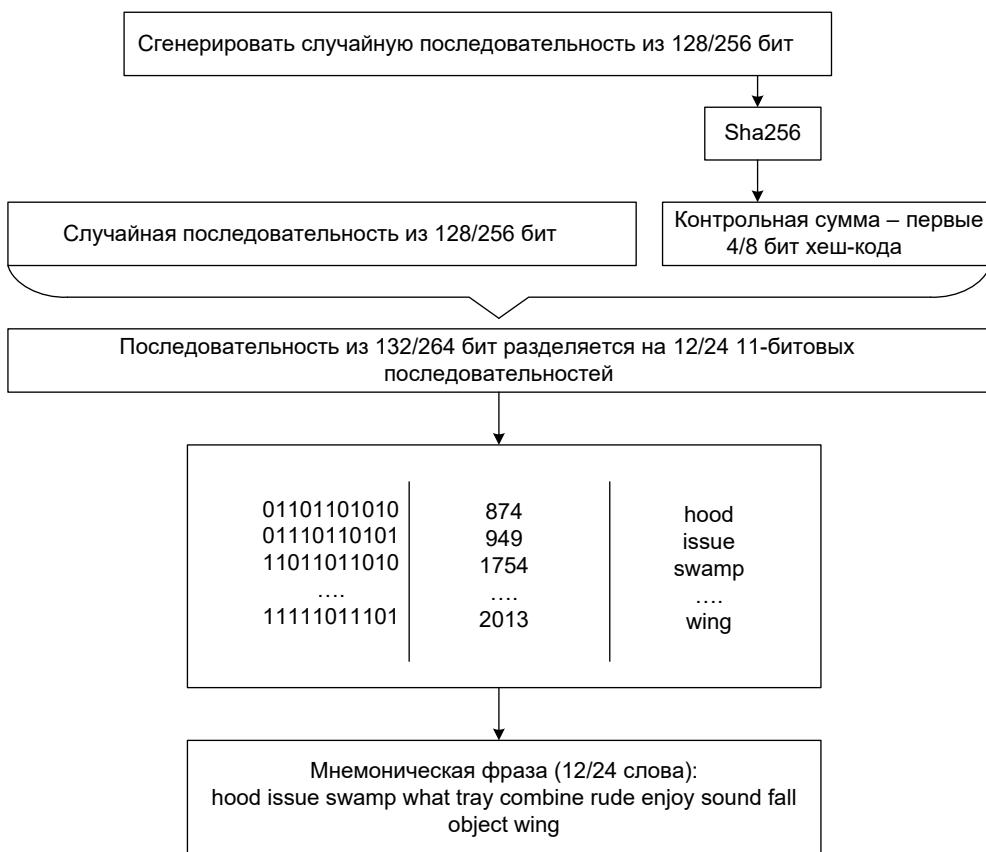


Рисунок 4.42. Преобразование случайной последовательности в мнемоническую фразу

Для того чтобы на практике опробовать работу мнемонического конвертора или убедиться в правильности работы вашего ПО, можно использовать онлайн-калькулятор Иана Колмана (Ian Coleman) [183].

Мнемоническая фраза используется для расширения ключа путем применения функции получения ключа на основе пароля семейства PBKDF2 (Password-Based Key Derivation Function 2).

Принцип действия данной функции прост и заключается в следующем: к мнемонической фразе добавляется соль, после чего к полученной последовательности применяется хеш-функция HMAC-SHA-512, в результате образуется 512-битовый хеш-код (сид-последовательность), из которого извлекается ключ (рис. 4.43) [48].

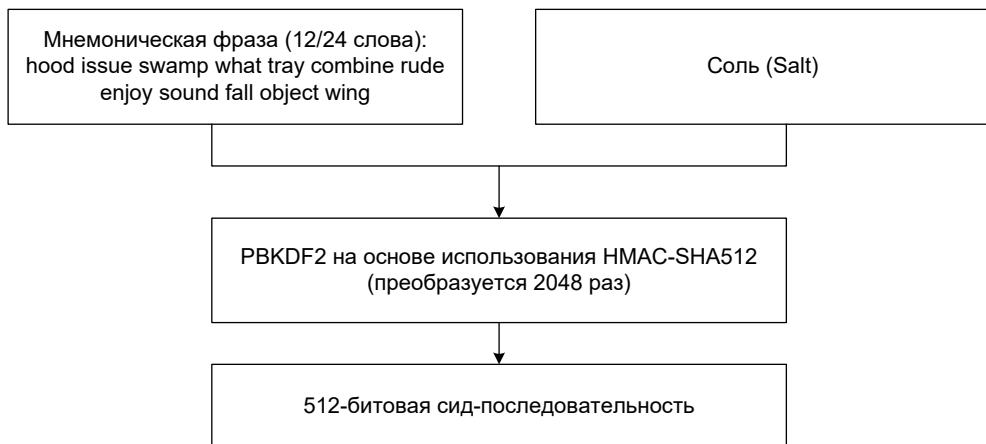


Рисунок 4.43. Расширение ключа с использованием мнемонической фразы

4.1.6 Создание и использование иерархических детерминированных ключей

Иерархический детерминированный протокол создания и передачи ключей (Hierarchical Deterministic protocol, протокол HD) значительно упрощает резервное копирование кошелька, устраняет необходимость в повторном обмене данными между несколькими программами, использующими один и тот же кошелек, позволяет создавать дочерние учетные записи, которые могут работать независимо, дает каждой родительской учетной записи возможность контролировать свои дочерние элементы, даже если дочерняя учетная запись взломана, и делит каждую учетную запись на части с полным и ограниченным доступом, чтобы ненадежным пользователям или программам было разрешено получать или отслеживать платежи, не имея возможности их тратить.

Протокол HD использует функцию создания открытого ключа ECDSA, point, которая принимает большое целое число (закрытый ключ) и превращает его в точку эллиптической кривой (открытый ключ):

public_key = point(private_key).

Благодаря принципу работы функции point (данная функция фактически выполняет скалярное умножение базовой точки эллиптической кривой на за-

крытым ключом) можно создать дочерний открытый ключ, объединив существующий (родительский) открытый ключ с другим открытым ключом, созданным из любого целочисленного значения i .

Этот дочерний открытый ключ является тем же самым открытым ключом, который был бы создан функцией `point`, если бы вы добавили значение i к исходному (родительскому) закрытому ключу по модулю q , который равен порядку простой подгруппы группы точек эллиптической кривой, и применили бы функцию `point` к полученному результату:

$$\text{point}((\text{parent_private_key} + i) \bmod q) == \text{parent_public_key} + \text{point}(i).$$

Это означает, что две или более независимые программы, которые соглашаются последовательность целых чисел, могут создать серию уникальных пар дочерних ключей из одной родительской пары ключей без дальнейшего взаимодействия. Более того, программа, которая распространяет новые открытые ключи для получения платежа, может делать это без какого-либо доступа к закрытым ключам, позволяя программе распространения открытых ключей работать на потенциально небезопасной платформе, такой как общедоступный веб-сервер.

Дочерние открытые ключи также могут создавать свои собственные дочерние открытые ключи, повторяя операции по получению дочерних ключей:

$$\text{point}((\text{child_private_key} + i) \bmod p) == \text{child_public_key} + \text{point}(i).$$

При создании дочерних открытых ключей или последующих открытых ключей предсказуемая последовательность целочисленных значений была бы не лучше, чем использование одного открытого ключа для всех транзакций, поскольку любой, кто знал один дочерний открытый ключ, мог найти все другие созданные дочерние открытые ключи из того же родительского открытого ключа. Вместо этого можно использовать случайное начальное число для детерминированной генерации последовательности целочисленных значений, чтобы связь между дочерними открытыми ключами была невидима для всех без знания этого начального числа.

Протокол HD использует одно корневое начальное число для создания иерархии дочерних и следующих за ними ключей с несвязанными детерминированно генерированными целыми числами. Каждый дочерний ключ также получает детерминированно генерированное начальное число от своего родителя, называемое чейнкодом, поэтому компрометация одного чейнкода не обязательно нарушает целочисленную последовательность для всей иерархии, позволяя главному чейнкоду продолжать быть полезным, даже если, например, взламывается веб-программа распространения открытых ключей.

На рис. 4.44 показан принцип работы протокола HD. Данный протокол использует следующие входные значения:

- родительский закрытый ключ и родительский открытый ключ – это обычные несжатые 256-битовые ключи ECDSA;
- родительский чейнкод – это 256 бит псевдослучайной последовательности;
- индекс – это 32-битовое целое число, определяемое программой.

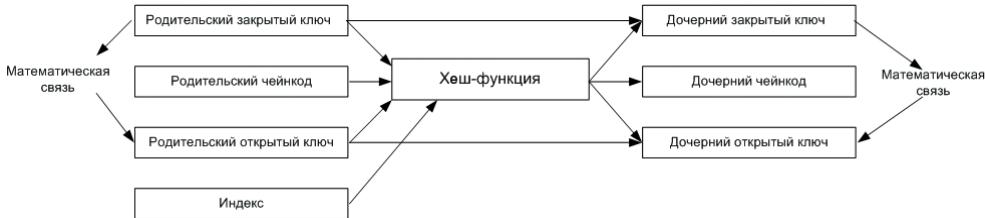


Рисунок 4.44. Вычисление дочерних ключей

Как показано на рис. 4.44, родительский чейнкод, родительский открытый ключ и индекс подаются на вход алгоритма хеширования HMAC-SHA-512 (см. главу 1). В результате работы данного алгоритма получается 512-битовое значение, которое используется следующим образом:

- старшие 256 бит используются как дочерний чейнкод;
- младшие 256 бит используются в качестве целочисленного значения, которое должно быть объединено либо с родительским закрытым ключом, либо с родительским открытым ключом, соответственно, для создания либо дочернего закрытого ключа, либо дочернего открытого ключа:

$\text{child_private_key} = \text{parent_private_key} + \text{lefthand_hash_output} \bmod p;$

$\text{child_public_key} = \text{point}(\text{parent_private_key} + \text{lefthand_hash_output} \bmod p)$

или

$\text{child_public_key} = \text{point}(\text{child_private_key}) = \text{parent_public_key} + \text{point}(\text{lefthand_hash_output}).$

Указание разных индексов приведет к созданию разных несвязанных дочерних ключей из одних и тех же родительских ключей. Повторение процедуры для дочерних ключей с использованием дочернего чейнкода создаст несвязанные ключи второго поколения.

Так как для создания дочерних ключей требуется как ключ, так и чейнкод, то ключ и чейнкод вместе называются расширенным ключом. Расширенный закрытый ключ и соответствующий ему расширенный открытый ключ имеют одинаковый чейнкод.

Родительский закрытый ключ и родительский чейнкод генерируются на основе случайных данных, как показано на рис 4.45.

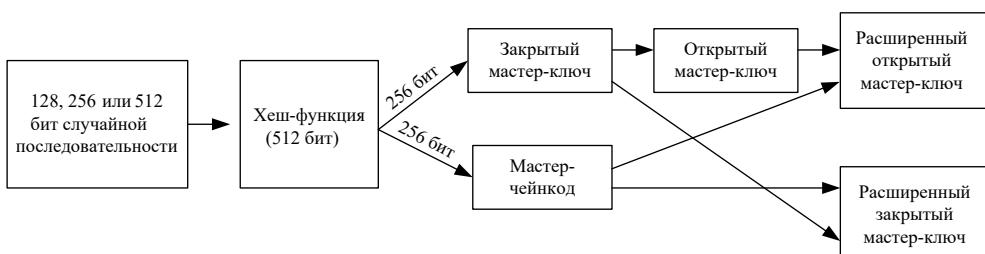


Рисунок 4.45. Создание мастер-ключей

Начальное инициализирующее значение, из которого в дальнейшем извлекается ключ, представляет собой 128, 256 или 512 бит случайных данных (в оригинальном описании системы Биткойн данное значение называется root seed, то есть корневое начальное число). Начальное инициализирующее значение – это единственные данные, которые необходимо сохранить пользователю, чтобы в дальнейшем иметь возможность извлечь ключи с использованием определенной программы-кошелька.

Как показано на рис. 4.45, начальное инициализирующее значение хешируется, в результате чего образуется хеш-код длиной 512 бит, из которого формируется закрытый мастер-ключ и мастер-чейнкод. Открытый мастер-ключ получается из закрытого мастер-ключа с помощью функции point. Полученный открытый мастер-ключ вместе с мастер-чейнкодом образуют расширенный открытый мастер-ключ.

4.2 Эфириум

После того как биткойн стал набирать обороты в своем развитии, его протоколы совершенствовались, появилась перспектива дальнейшего развития блокчейн-технологий. Так, в 2015 году в свет вышла новая криптовалютная система Эфириум (Ethereum), продемонстрировавшая концептуально новый подход к построению систем распределенных реестров.

Новизна заключалась в том, что теперь стало возможно дополнять транзакции вызовами различных функций. Начался период блокчейн-платформ со смарт-контрактами.

В общем случае модель системы Эфириум можно представить следующим образом:

- децентрализованные приложения;
- смарт-контракты, благодаря которым пользователи разрабатывают различные децентрализованные приложения;
- виртуальная машина (EVM) хранит в себе глобальное состояние системы, внутри которой циркулируют смарт-контракты;
- ноды (узлы), взаимодействуя между собой, формируют виртуальную машину;
- интернет, в котором существует несколько тысяч нод.

4.2.1 Глобальное состояние

В отличие от системы Биткойн, которая не имеет как такового зафиксированного состояния в какой-либо момент времени (и, в частности, где баланс пользователей определяется путем пересчета количества не израсходованных ими выходов), подход к отслеживанию состояния системы платформы Эфириум осуществляется принципиально иначе.

Одним из основных понятий в системе является ее глобальное состояние (global state). Начиная с генезис-блока, каждый блок хранит в себе информацию о состоянии системы в момент выработки этого блока – хеш-код глобального состояния.

По мере создания транзакций и выработки новых блоков количество пользователей и их балансы в системе, а также количество опубликованных

в сети контрактов изменяются. Это учитывается при добавлении новых блоков в цепочку.

Структура глобального состояния представляет из себя префиксное дерево (Merkle Patricia tree), которое несколько отличается от обычного бинарного дерева Меркля. Префиксное дерево позволяет динамически добавлять информацию в его различные ветви, обозначая изменение состояния, но, как и с бинарным деревом, для его пересчета и проверки целостности понадобится пересчитать только интересующую ветвь [16].

Глобальное состояние можно представить, как показано на рис. 4.46.

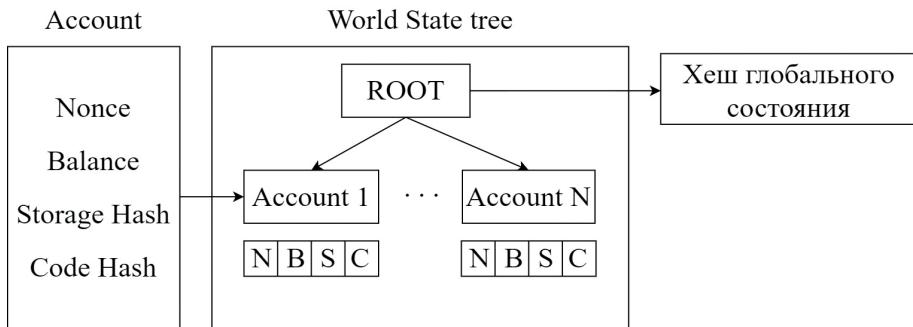


Рисунок 4.46. Глобальное состояние сети Эфириум

При его формировании учитывается текущее состояние всех адресов и аккаунтов в сети. Учитываются следующие данные для каждого пользователя:

- Nonce – количество опубликованных пользователем контрактов;
- Balance – текущий баланс пользователя;
- Storage Hash – хеш-код всей информации об указанном адресе;
- Code Hash – хеш-код программного кода смарт-контракта, связанного с данным адресом.

Когда транзакции изменяют состояние аккаунтов, изменяется и глобальное состояние, его новый хеш-код добавляется в блок.

4.2.2 Консенсус

Ранее мы уже рассказывали о различных протоколах консенсуса в системах распределенного реестра (см. главу 3). Теперь мы постараемся немного подробнее описать адаптированный вариант консенсуса на основе доказательства работы PoW, используемый в системе Эфириум.

В основе механизма консенсуса PoW лежит майнинг. В сети Эфириум для поддержания работы механизма консенсуса PoW используется алгоритм Ethash [3]. Данный алгоритм был разработан для того, чтобы предотвратить майнинг с помощью устройств на основе специализированных микросхем (ASIC-устройства).

Ethash предполагает использование DAG-файла (на основе направленного ациклического графа), который со временем увеличивается в размере, на 8 Мб каждые 3000 блоков, представляющие так называемую «эпоху». Такой механизм не позволит майнить на ASIC-устройствах, вынуждая использовать только графические процессоры, обладающие достаточной памятью для хранения DAG-файла.

В настоящий момент размер файла преодолел порог в 4 Гб и целый ряд графических вычислителей оказался неактуальным в задаче формирования блока.

Таким образом, процесс майнинга на платформе Эфириум несколько отличается от типичного алгоритма в сети Биткойн, который был описан в главе 4 ранее.

Генерирующий блок узел выбирает случайное число *nonce* и вместе с предобработанным заголовком предыдущего блока передает его на вход функции хеширования Keccak-256 (см. главу 1), полученный результат указывает на 128-байтовый сектор в DAG-файле. После этого хеш-код и выбранный сектор смешиваются с помощью специальной функции *mix*, полученный результат укажет на новый сектор DAG-файла. Данная операция выполняется 64 раза, после чего результат сжимается до 32 байт.

Полученное значение *mixed_hash* сравнивается с заданным значением сложности. Если оно больше порогового значения сложности, то выбирается новое значение *nonce*, случайно или путем инкрементирования [31]. Если полученный результат удовлетворяет условию, то значение *nonce* для данного блока определено, доказательство работы выполнено и будет легко проверено остальными участниками сети.

В ходе работы системы значение сложности вычисляется для каждого блока заново по определенной формуле, зависящей от различных параметров, в том числе от времени выработки предыдущего блока. Так, например, если текущий блок был выработан намного быстрее, чем предыдущий, то для следующего блока необходимо будет увеличить значение сложности, и наоборот. С помощью такого механизма время выработки блока стабилизируется и приводится к среднему значению порядка 15 секунд.

На основе алгоритма Ethash работает не только платформа Эфириум, но и альтернативные системы, такие как Ethereum Classic, Ubiq, Expanse.

4.2.3 Газ

Для формирования распределенной системы, которая бы умела работать со смарт-контрактами, т. е. могла бы формировать транзакции с вызовами функций кода, написанного на Тьюринг-полном языке, понадобилось обезопасить систему от различного типа попыток «заспамить» систему, экстренно увеличить в ней нагрузку и вывести из строя.

Для этого разработчиками была создана система оценки нагрузки каждой транзакции на узлы системы. Подход рассматривает каждую транзакцию как набор элементарных операций, которые могут быть выполнены на узле при проверке транзакции в момент формирования блока. При этом каждая операция стала стоить определенное количество так называемого «газа» (gas).

Теперь перед пользователем будет стоять требование предоплаты вызовов функций – обеспечить суммарное количество газа, необходимое для выполнения транзакции [137].

При формировании вызова пользователь указывает два параметра – GasLimit и GasPrice. В случае если при проверке транзакции возникнет непредвиденное или целенаправленное сверхвысокое количество операций, то их выполнение будет прервано в момент, когда это число превысит максимальное разрешенное пользователем значение GasLimit.

В значении GasPrice отправитель транзакции указывает стоимость одной единицы газа в wei (wei – дробная единица основной криптовалюты системы Эфи-

риум – эфира (ether или Eth); 1 вей равен 10^{-18} эфира). Чем это значение выше, тем более привлекательна данная транзакция будет для майнеров и тем быстрее она будет обработана и добавлена в блок – за каждую транзакцию, добавленную в блок, майнеры получают дополнительное вознаграждение, комиссию.

Значение *GasPrice* * *GasLimit* будет характеризовать максимальную прибыль от этой транзакции, однако размер комиссии будет высчитываться исходя из реально потребленного количества газа [50].

4.2.4 Адреса и кошельки

В системе Эфириум у каждого пользователя существует его уникальный идентификатор – его адрес в сети. Этот адрес будет характеризовать отправителя транзакции, а также использоваться для перевода средств. Пользователь может обладать несколькими адресами, переводить валюту с одного счета на другой, пользоваться каждым для отдельных целей и т. д.

Как и в системе Биткойн, адрес пользователя вычисляется из его открытого ключа; в качестве адреса используются последние 20 байт результата хеширования открытого ключа пользователя алгоритмом хеширования Кессак-256.

Кошельком в данном случае называют программное обеспечение, которое помогает пользователю удобно одновременно управлять несколькими счетами. Кошелек защищен паролем пользователя и сохраняет счета пользователей в безопасности. Одними из наиболее популярных кошельков являются официальный кошелек системы MyEtherWallet, приложение Mist, мобильный кошелек jaxx и др.

Для работы с сетью пользователю необходимо располагать парой ключей – закрытым и открытым. Пара ключей вырабатывается в соответствии с алгоритмом ECDSA на основе эллиптической кривой secp256k1 (см. главу 2). Открытый ключ пользователя хешируется алгоритмом Кессак-256, после чего от результирующего хеш-кода берутся самые правые 20 байт, т. е. 40 шестнадцатеричных символов, которые и будут являться идентификатором пользователя.

В момент создания закрытый ключ шифруется с помощью пароля пользователя с использованием симметричного алгоритма AES-256 [131] и сохраняется на устройстве пользователя в зашифрованном виде.

Чтобы управлять своим счетом, пользователю необходимо располагать на устройстве файлом со своим закрытым ключом, а также помнить пароль, для того чтобы расшифровать его.

Помимо адресов пользователей, в системе присутствует не менее важный набор адресов – адреса смарт-контрактов (каждый контракт, как и пользователь, обладает адресом, но, в отличие от пользователя, у контракта нет собственного закрытого ключа). В следующем разделе мы напишем подробнее о транзакциях создания контракта. В этот момент генерируется новый адрес, который устанавливается в ассоциацию с исполняемым кодом.

По аналогии с адресом пользователя адрес контракта обладает собственным балансом. Это в том числе позволяет расширить функционал смарт-контрактов. В остальном эти адреса принципиально различаются. Основным отличием пользовательского адреса от адреса контракта является возможность формирования транзакций [1]. Логично, что пользовательский адрес может являться инициатором транзакций, контракт же, в свою очередь, выступает только исполнителем и не может самостоятельно без внешнего вызова исполнять те или иные команды.

4.2.5 Транзакции

Транзакции сети Эфириум можно разделить на два ключевых типа:

- транзакции вызова – включают в себя перевод криптовалюты от одного пользователя другому или вызов функции уже опубликованного смарт-контракта;
- транзакции создания смарт-контракта в сети – каждая такая транзакция создает новую учетную запись и ассоциацию с ней программного кода; подобные транзакции несут в себе несколько больше информации.

Для обоих типов транзакций есть целый ряд общих параметров:

- Nonce – количество транзакций, сформированное данным пользователем; поле должно являться уникальным и будет идентифицировать транзакцию;
- GasPrice – указывает количество wei – стоимость, которую пользователю необходимо будет заплатить за каждую единицу газа при выполнении транзакции;
- GasLimit – предельное количество газа, которое пользователь готов заплатить за выполнение данной транзакции; в случае если выполнение транзакции, например выполнение функции смарт-контракта, будет требовать газа больше, чем указано в параметре GasLimit, то выполнение транзакции прервется, и состояние системы вернется к тому, которое было до начала выполнения данной транзакции;
- From – отправитель данной транзакции;
- Signature – электронная подпись транзакции, вычисленная по алгоритму ECDSA с использованием кривой secp256k1 и хеш-функции Кессак-256.

Помимо этих параметров, каждый тип транзакции обладает собственным набором информативных полей. В частности, для транзакций вызова указывается следующая информация:

- To – учетная запись, в отношении которой совершается транзакция, например пользователь-получатель перевода, либо смарт-контракт, чья функция была вызвана данной транзакцией;
- Value – количество криптовалюты в wei, которое передается вместе с выполнением транзакции, например сумма перевода или необходимая сумма для выполнения функции смарт-контракта;
- InputData – входные параметры вызова, например параметры для функции смарт-контракта.

4.2.6 Структура блока

Рассмотрим параметры блоков в сети Эфириум. Выше мы описали основной набор информации, который циркулирует в сети, какие данные хранятся в блоках и для чего они предназначены, теперь же попробуем сделать некоторое обобщение и структурируем информацию.

Стандартный набор данных блока включает в себя следующее:

- Block Height (высота блока) – высотой принято называть текущую длину цепочки, то есть количество блоков в ней;
- Timestamp – временная метка создания блока;

- Transactions – количество транзакций, добавленных в данный блок;
- Miner (получатель награды) – адрес пользователя в сети Эфириум, который создал блок и получил вознаграждение за него, а также комиссию от транзакций – 160 бит, 40 шестнадцатеричных символов;
- Reward – награда, которую получит майнер за создание блока, – на текущий момент стандартная награда составляет 2 эфира за выработку блока, а также к этому значению добавляется сумма комиссий за транзакции в блоке;
- Difficulty – сложность, определенная для данного блока;
- Size – размер блока в байтах;
- Gas Used – суммарное количество газа, израсходованное всеми транзакциями блока;
- Gas Limit – установленное ограничение на количество газа, суммарно потребляемое транзакциями данного блока;
- Extra Data – дополнительные данные, которые могут быть внесены создателем блока.

Дополнительная информация, включаемая в блок:

- Hash – хеш-код данного блока;
- Parent Hash (родительский хеш) – хеш-код заголовка предыдущего блока;
- Ommers Hash / Sha3 Uncles (хеш дядей) – хеш-код списка «дядей» (описаны далее) для данного блока;
- State Root – корень состояния; ранее мы описывали построение дерева состояния, в каждый блок включается хеш-код дерева в текущий момент, после того как все транзакции будут добавлены в глобальное состояние;
- Transactions Root – хеш-код корня дерева всех транзакций, включенных в данный блок;
- Receipts Root – корень дерева квитанций; для каждой транзакции после ее успешного выполнения создается квитанция, в этой квитанции отображается, в какой блок была добавлена транзакция, хеш-код этой транзакции, какое количество газа она потребляет, какое количество потребляют суммарно все транзакции, которые уже добавлены в блок, и т. д.;
- Logs Bloom – журнал, содержащий различную служебную информацию в форме логов;
- Nonce – случайное число, используемое алгоритмом Ethash для выработки хеш-кода текущего блока согласно консенсусу PoW;
- MixedHash – вместе со значением Nonce может использоваться для проверки выполнения условия консенсуса PoW.

4.2.7 Эволюция системы Эфириум

Ранее, в главе 3, было рассмотрено понятие форков. Описывались ситуации с разветвлением системы в тех или иных случаях.

Для платформы Эфириум форки также актуальны. Обычные форки цепочки порождают ommits, или так называемых «дядей», которые были упомянуты при описании структуры блока. Дядями называют устаревшие блоки более коротких цепочек, однако такие блоки не просто отбрасываются – в цепочку транзакции этих блоков не попадают, однако их владельцам назначается не-

большое вознаграждение. В результате этого у майнеров появляется стимул работать над обеспечением целостности цепочки, формировать конкуренцию между пользователями за основное вознаграждение и комиссии с транзакций.

Гораздо более важными являются хардфорки и softфорки системы Эфириум. Давайте проследим их историю, узнаем, в чем была причина основных из них и к чему это привело.

Frontier – «Рубеж» (блок № 1)

Цепочка была запущена 30 июля 2015 года, эту стадию принято называть Frontier. Дальнейшая жизнь платформы видоизменялась от форка к форку. Преобразования были как запланированными, так и вынужденными.

Спустя полтора месяца, 14 марта 2016 года, по достижении длины цепочки в 200 000 блоков был выпущен пакет обновлений системы Ice Age («Ледниковый период»). Это было вызвано необходимостью усовершенствования безопасности системы и скорости ее работы.

Homestead – «Усадьба» (блок № 1 150 000)

14 марта 2016 года, спустя год после запуска сети, команда разработчиков сделала решительный шаг – выпустила новую версию программного обеспечения, заявив о завершении тестирования сети и готовности к стабильной работе. Этот релиз получил название Homestead и был активирован на блоке с номером 1 150 000.

DAO Fork (блок № 1 920 000)

Использование смарт-контрактов набирало популярность, все больше пользователей становились владельцами цифровых активов. Компании стали выпускать свои токены, вести бизнес с помощью криптовалютной площадки.

Наиболее популярной децентрализованной организацией в сети стала краудфандинговая компания The DAO. Однако свою известность компания приобрела в ходе крупнейшего скандала за всю историю системы Эфириум [120].

Все дело в том, что злоумышленники нашли в используемом The DAO смартконтракте возможность неограниченного вывода средств на посторонний адрес. Так, почти треть активов компании, по курсу того времени – 50 миллионов долларов, была украдена злоумышленником, чьему никак не могла помешать система, ведь контракт был выполнен верно, а вина остается за его разработчиком.

Пытаясь найти компромисс в сложившейся ситуации, создатель сети Виталик Бутерин (Vitalik Buterin) предложил сформировать новый форк, разделив сеть за некоторое время до кражи, чтобы транзакции злоумышленников были забыты. Этот вариант решения выглядел одним из наиболее оптимальных, но он стал причиной огромных разногласий и идеологических споров.

С одной стороны, необходимо было вернуть доверие к площадке и устранить действия хакеров, с другой – идеологически это нарушало принцип децентрализации сети, отсутствия в ней централизованного механизма управления, ведь контракт выполнил те действия, которые ему были предписаны.

В результате 20 июля 2016 г. еще совсем молодая система, оказавшись в столь сложном положении, потеряла часть своих пользователей. Форк действительно был создан, но его противники продолжили пользоваться оригинальным про-

должением цепочки. Сегодня мы знаем эту систему как Ethereum Classic. Курс ее криптовалюты заметно меньше курса официальной площадки, большой популярностью она не пользуется, в связи с чем и уступает по развитию своему «братью».

Metropolis: Byzantium – «Византия» (блок № 4 370 000)

Еще в 2015 году были анонсированы планы по развитию проекта Эфириум. Один из крупнейших апгрейдов был обозначен как Metropolis и включал в себя несколько фаз развития.

Первой фазой стал софтфорк Byzantium, произошедший 16 октября 2017 года. Форк предлагал обновление программного обеспечения платформы, которое оптимизировало работу системы, увеличило скорость обработки транзакций, а также сократило награду майнерам с пяти до трех Eth; был обозначен задел на использование криптографических протоколов на основе доказательства с нулевым разглашением zk-SNARK.

Metropolis: Constantinople – «Константинополь» (блок № 7 280 000)

Столь долгожданное новое обновление платформы было назначено на блок № 7 080 000 и было выполнено в Ropsten – тестовой сети Эфириум.

Обновление несло новые изменения, улучшающие масштабируемость системы, было уменьшено в 10 раз потребление газа, изменена его стоимость, награда майнеру сократилась до 2 Eth. Кроме того, в Constantinople заложены ключевые изменения в механизмы платформы, которые стали первыми шагами к переходу от консенсуса PoW к PoS, сделав консенсус системы на некоторое время их гибридной версией.

Новое обновление, как всегда, было встречено пристальным вниманием, в том числе со стороны компаний, контролирующих информационную безопасность и проводящих аудит информационных систем. Своевременно была обнаружена критическая уязвимость, благодаря ей злоумышленники могли бы получить доступ к счету любого пользователя [118]. Уязвимость быстро закрыли с помощью форка St. Petersburg и уже на высоте 7 280 000 блока 28 февраля 2019 года оба обновления были добавлены в один и тот же блок настоящей цепи, не давая воспользоваться существовавшей уязвимостью.

Стоит также заметить, что это был форк из числа хардфорков, по сути система вновь расслоилась пополам. Всем пользователям, желающим перейти на новую версию форка, необходимо было заранее обновить программное обеспечение, до момента формирования блока с «имплементацией», встраиванием новых параметров системы. В противном случае пользователи могли либо навсегда остаться в старой версии системы, либо потерять часть средств, которыми успели воспользоваться после выхода обновления.

Istanbul – «Стамбул» (блок № 9 069 000)

Сеть получила новое обновление зимой 2019 года. Новшества вновь направлены на поддержку масштабирования, а также на сопротивление DoS-атакам. На платформе появились все необходимые инструменты для работы с технологией zk-SNARK и взаимодействия с криптовалютой Zcash, предпосылки которых появились еще в Constantinople. Платформа продолжает движение к консенсусу PoS [7].

Berlin – «Берлин» (блок № 12 224 000)

Форк состоялся 15 апреля 2021 г. и включает следующие основные обновления [42]:

- оптимизация оценки газа для операций возведения в степень по модулю и, как следствие, уменьшение стоимости выполнения ряда криптографических алгоритмов (например, проверки подписи RSA и выполнения некоторых других функций);
- введен новый тип транзакций – «транзакции-оболочки» (Typed Transaction Envelope), который делает все будущие типы транзакций обратно совместимыми благодаря возможности «запаковывать» в транзакции-оболочки другие транзакции, типы которых планируется добавить в будущем;
- увеличена стоимость (до трех раз) использования некоторых кодов операций, что защищает от атак типа «отказ в обслуживании» и повышает скорость обработки транзакций.

London – «Лондон» (блок № 12 965 000)

Вышел 5 августа 2021 г.; основные изменения состоят в следующем:

- изменен принцип формирования комиссий за транзакции: от фактически аукционной формы произошел переход к вычисляемым комиссиям, что должно снизить расходы на проведение транзакций;
- отложен запланированный ранее (и внедренный в систему Эфириум) переход на консенсус PoS в рамках Ethereum 2.0.

Serenity – «Безмятежность» – Ethereum 2.0

В апреле 2020 года в тестовой сети совершенно нового Ethereum 2.0 Topaz был сформирован первый блок новой цепочки. Позже, 1 декабря 2020 г., был выпущен genesis-блок оригинальной сети Ethereum 2.0 [39].

Сейчас сеть Ethereum 2.0 проходит различные тестирования, но в ближайшей перспективе планируется полный перевод пользователей и их активов на новую площадку. До этого времени система второй версии будет играть лишь тестовую роль.

Предложения по улучшению системы Эфириум

Немаловажное значение в эволюции системы Эфириум играют и предложения по ее улучшению – EIP (Ethereum Improvement Proposals). По аналогии с BIP для системы Биткойн периодически предлагаются новые улучшения системы Эфириум EIP, и некоторые из них добавляются в цепочку, в частности [119]:

- EIP 1234 – вознаграждение для майнеров сокращается с трех до двух Eth [221];
- EIP 145 – эффективность и скорость блокчейна увеличивается при помощи добавления переключающих механизмов к виртуальной машине Эфириум [65];
- EIP 1052 – сеть начинает потреблять меньше энергии [151];
- EIP 1283 – снижается потребность в использовании внутренней валюты блокчейна Эфириум – газа [233].

4.2.8 Основная и тестовые сети платформы Эфириум

Как правило, все действия, транзакции, вызовы функций смарт-контрактов выполняются в основной сети платформы (Mainnet). Система работает по всем правилам, которые были описаны ранее. Курс криптовалюты применим именно для нее, все токены и активы имеют ценность именно в основной цепочке.

Однако, помимо основной цепочки, существует несколько альтернативных ее версий. Такие цепочки называются тестовыми (*testnets*) и, как следует из названия, применяются для тестирования разработчиками различных механизмов сети, отладки смарт-контрактов и т. д. Такие тестовые сети являются глобальными и доступны всем пользователям интернета.

Кроме тестовых сетей, каждому разработчику доступно создание своей собственной частной, или приватной, сети. Эта сеть будет строиться на кастомизированных разработчиком параметрах. Будут заданы необходимые настройки, что опять же позволяет протестировать различные приложения в тех или иных условиях работы системы. Такие сети чаще доступны в рамках локальных сетей.

Каждая сеть характеризуется ее идентификатором (*network ID*). Так, например, основная сеть Эфириум имеет идентификатор 1, Ethereum Classic – 61. Приватная сеть также должна иметь свой идентификатор, при запуске необходимо его указать.

Идентификатор не должен совпадать с идентификаторами других глобальных сетей. Со списком основных и тестовых глобальных сетей можно ознакомиться на сайте chainlist.org [94].

Наиболее популярными глобальными тестовыми сетями можно считать следующие:

- Ropsten Testnet (*ID* = 3) – сеть была запущена в 2016 году, позволяет пользователям тестировать свои приложения и смарт-контракты [216]. Максимально полно дублирует механизмы текущего состояния основной сети, в том числе гибридный консенсус, время выработки блока и др. В 2017 году сеть была подвержена спам-атаке, в связи с чем вышла из строя на короткий срок, после чего была восстановлена с более безопасными параметрами;
- Rinkeby Testnet (*ID* = 4) – вместо PoW-консенсуса, применяемого основной сетью и сетью Ropsten, использует вариант консенсуса PoA. Это позволило сделать инструмент устойчивым к практически любого рода спам-атакам [208].

При работе с тестовыми сетями возникает вопрос: как выполнять операции в системе, ведь выполнение транзакции также требует оплаты комиссии, газа, потраченного на выполнение команд. Вариант неограниченного баланса не подошел бы в таких сетях, и системы постоянно были бы атакованы.

Для решения этого вопроса в тестовых системах существуют так называемые «краны» (*faucets*) – сервисы, периодически предоставляющие тестовые средства на указанный кошелек в сети. Так, например, кран сети Ropsten [215] один раз в день предоставляет тестовые 0,3 Eth на уникальный IP-адрес.

4.2.9 Запуск сети Эфириум

Нужно понимать, что Эфириум – это глобальная платформа, которой пользуется большое количество пользователей. Конечно, можно использовать основную сеть Эфириум, в которой происходят все транзакции. Но при разработке и тестировании смарт-контрактов используются специальные тестовые сети, в том числе рассмотренные в предыдущем разделе.

Рассмотрим основные шаги, которые следует выполнить для начала работы с платформой Эфириум.

MetaMask

MetaMask [179] представляет собой криптовалютный кошелек, который устанавливается как расширение к браузеру. Он подходит как для работы с глобальной сетью, так и для работы с приватными сетями, тестовыми и частными.

Если вы первый раз создаете криптовалютный кошелек, то он предлагает вам сервис для хранения и управления несколькими адресами (рис. 4.47). Например, у вас есть 5 адресов, тогда вы можете их хранить и распоряжаться средствами в одном кошельке.

Впервые в MetaMask?

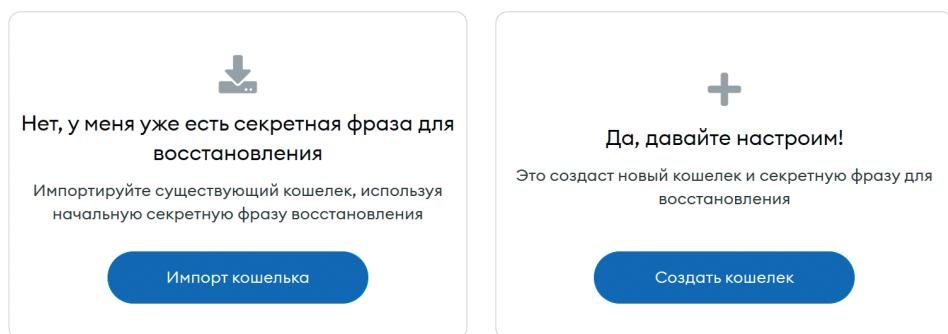


Рисунок 4.47. Установка криптовалютного кошелька MetaMask

Для того чтобы создать свой кошелек, необходимо ввести личные данные, которые запрашивает расширение, после чего появится окошко с секретной резервной фразой (secret recovery phrase), пример которого приведен на рис. 4.48.

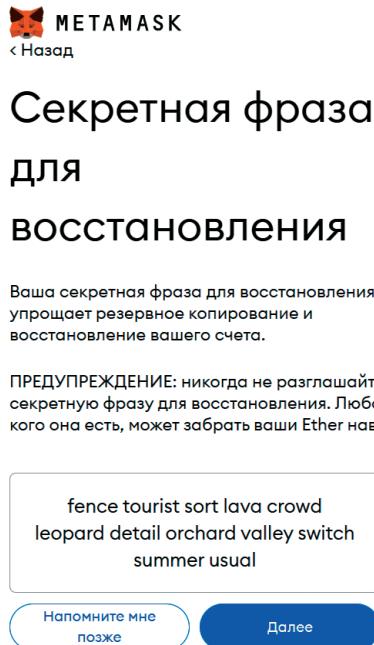


Рисунок 4.48. Выбор секретной фразы кошелька MetaMask

Секретная фраза позволяет выработать из случайного набора слов средство восстановления ключа. Ее необходимо сохранить в удобное и защищенное место. Далее платформа просит ввести заданную фразу, чтобы убедиться, что пользователь ее запомнил.

После успешного входа можно приступать к работе с криптовалютным кошельком. Если вы хотите работать с сетью Mainnet – основной сетью платформы, то необходимо остаться в том окне, которое было загружено при входе. Мы же в качестве примера приведем работу с тестовой сетью.

MetaMask по умолчанию поддерживает четыре тестовые сети:

- Ropsten;
- Kovan;
- Rinkeby;
- Goerli.

Будем работать с тестовой сетью Ropsten. Для этого необходимо в правом верхнем углу в списке сетей выбрать сеть Ropsten. Однако при этом визуально ничего не изменится: аккаунт и адрес остаются прежними.

Преимущество тестовой сети заключается в возможности получить тестовый эфир и с помощью полученных средств заниматься тестированием различных приложений, которые будут основаны на платформе Эфириум.

Перейдем на сайт Ropsten Faucet [215]. Как было сказано выше, это кран – сервис, подвязанный к сети Ropsten, который выдает тестовые средства (рис. 4.49). Нужно в появившемся окне указать адрес тестового кошелька и нажать на кнопку «Отправить тестовый Эфир» (Give me Ropsten ETH!).

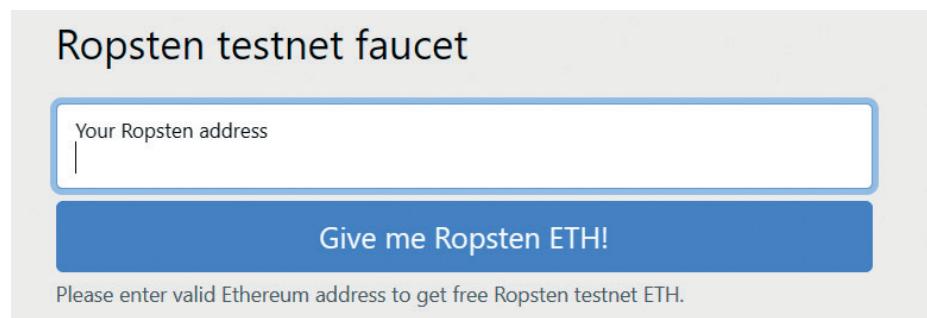


Рисунок 4.49. Получение эфира в тестовой сети

После нажатия запрос будет добавлен в очередь. Чтобы посмотреть информацию о зачисленных тестовых средствах, необходимо в кошельке MetaMask нажать на меню (три вертикальные точки) и выбрать действие «Посмотреть на Etherscan» (View Account on Etherscan) (рис. 4.50).

Txn Hash	Method	Block	Age	From	To	Value	Txn Fee
0xd6b67854ebafc0e1c...	Transfer	12459346	21 secs ago	0x49228b1075578139fc...	OUT 0x373b9b6b6f2dcfc9126...	0.3 Ether	0.0000315

Рисунок 4.50. Просмотр информации о тестовых средствах

Etherscan [235] – один из сервисов, позволяющих отслеживать транзакции и просматривать всю информацию о событиях, происходящих в сети Эфириума. Можно просмотреть информацию по конкретному адресу, блоку или транзакции. Важно отметить, что получать тестовые средства можно только 1 раз в сутки по одному IP-адресу.

На главной панели Etherscan можно увидеть заданный адрес, баланс счета, транзакции и расширенную информацию (см. рис. 4.50).

В блоке с транзакциями указаны адреса отправителя и получателя, количество эфира и налог за перевод средств. В тестовой сети соблюдаются все те же правила, что и в главной сети.

Вернувшись к кошельку MetaMask, можно увидеть, что тестовый баланс обновился и составляет 0,3 Eth. Теперь можно оперировать полученным активом. В частности, можно подключить USB-устройство для сохранения ключей, открыть новый счет и переводить деньги между счетами.

Однако при переводе между счетами будет сформирована транзакция, и сеть будет просить комиссию за перевод так же, как если бы это было в главной сети. Комиссия при этом будет начисляться на счет майнера. При переводе

средств можно менять скорость транзакции, что скажется на размере комиссии (меньше скорость – меньше комиссия).

Давайте создадим новый счет и попробуем перевести на него часть средств. Для этого необходимо в правом верхнем углу нажать на управление аккаунтом (кружочек) и выбрать действие «Создать счет» (рис. 4.51). Создадим «Счет 2».

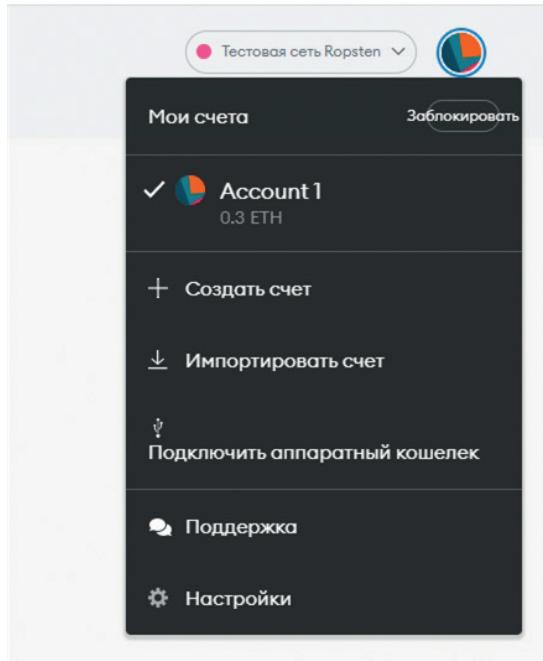


Рисунок 4.51. Создание нового счета

Теперь у нас есть два счета, и мы можем с одного счета перевести на другой, например, 0,1 Eth. При выборе соответствующего действия появляется окошко с подтверждением транзакции, на котором мы видим (рис. 4.52):

- счет отправителя и счет получателя;
- количество средств, которое мы хотим перевести;
- количество газа (комиссия за перевод);
- скорость транзакции;
- итог.

Скорость транзакции можно изменить, изменится и количество газа (меньше скорость сделки – меньше газа придется заплатить отправителю).

Когда выполняется перевод между своими счетами, со счета отправки списывается также комиссия за перевод.

Кроме глобальных тестовых сетей, которыми пользуются пользователи со всего мира, существует возможность организовать свою частную тестовую сеть.

Для этого существуют различные средства и программное обеспечение, о которых речь пойдет далее.

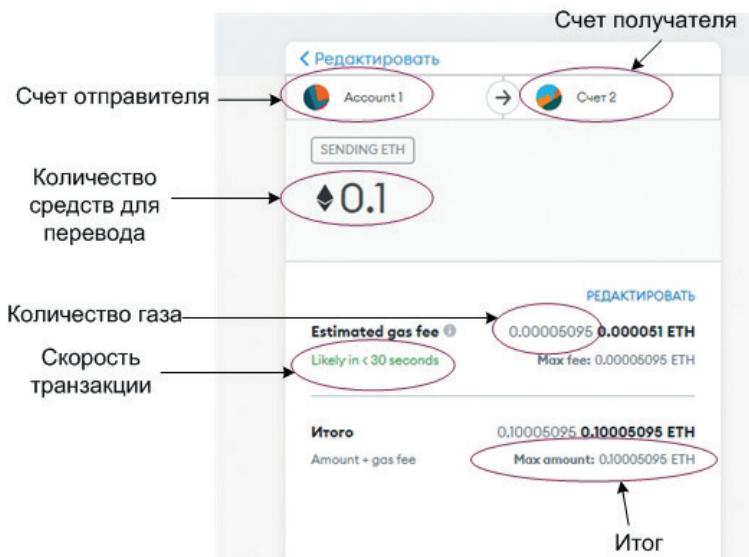


Рисунок 4.52. Перевод средств между двумя счетами

Ganache

Ganache является одним из эмуляторов сети, построенной на платформе Эфириум. Это свободно распространяемое ПО, доступное для скачивания [136].

Для того чтобы начать работу с этой программой, создадим новую сеть, щелкнув по кнопке «New Workspace» (рис. 4.53).

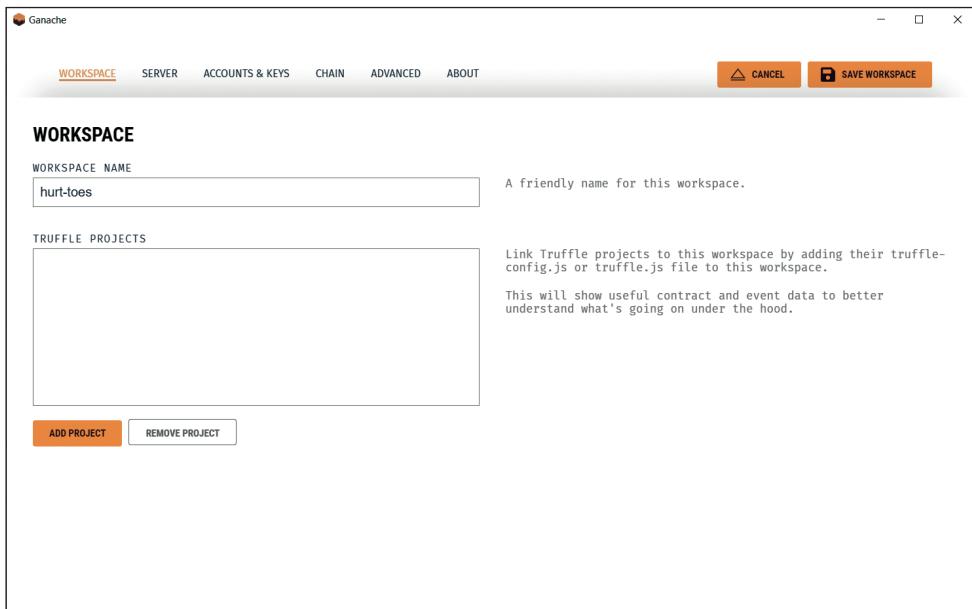


Рисунок 4.53. Главное окно Ganache

В открывшемся окне можно указать название проекта и подключить сторонние проекты сервиса Truffle (trufflesuite.com), если вы создавали проекты на этой платформе.

На вкладке «Server» находятся настройки сервера (рис. 4.54). По умолчанию заданы следующие параметры:

- «Hostname»: 127.0.0.1 – Loopback Pseudo-Interface 1;
- «Port number»: 7545.

Для параметра «Network ID» необходимо указать какой-нибудь ID сети, для того чтобы сети отличались друг от друга. У любой сети (тестовой или глобальной) тоже есть свои идентификаторы, поэтому рекомендуется поставить значение больше 0010.

Также на вкладке «Server» оставляем включенными переключатели «Automine» и «Error on transactions failure» и выключенным переключатель «Chain forking».

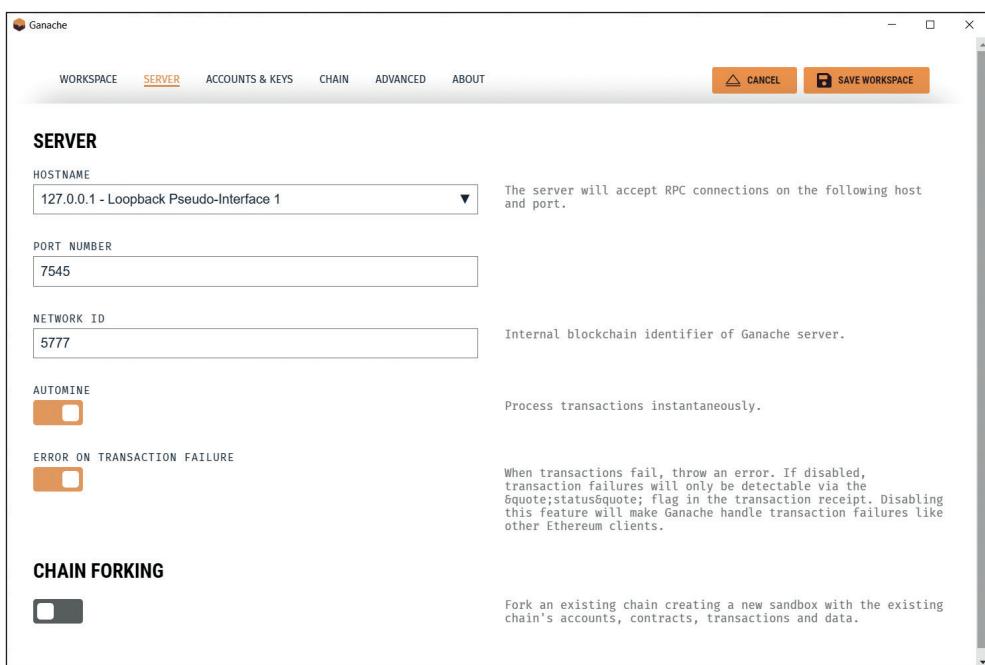


Рисунок 4.54. Вкладка «Server»

На вкладке «Accounts and Keys» (рис. 4.55) можно указать, какое количество адресов с каким количеством средств должно быть создано в сети.

По умолчанию установлен баланс для всех создаваемых пользователей в поле «Account default balance», равный 100 Eth. Также по умолчанию в поле «Total accounts to generate» задано 10 аккаунтов.

Переключатель «Autogenerate HD mnemonic» отвечает за генерацию мемориической фразы. Его можно отключить, тогда будет использоваться фраза, указанная ниже, в поле. Переключатель «Lock accounts» также оставим выключенным. Необходимость блокировки аккаунтов будет рассмотрена далее.

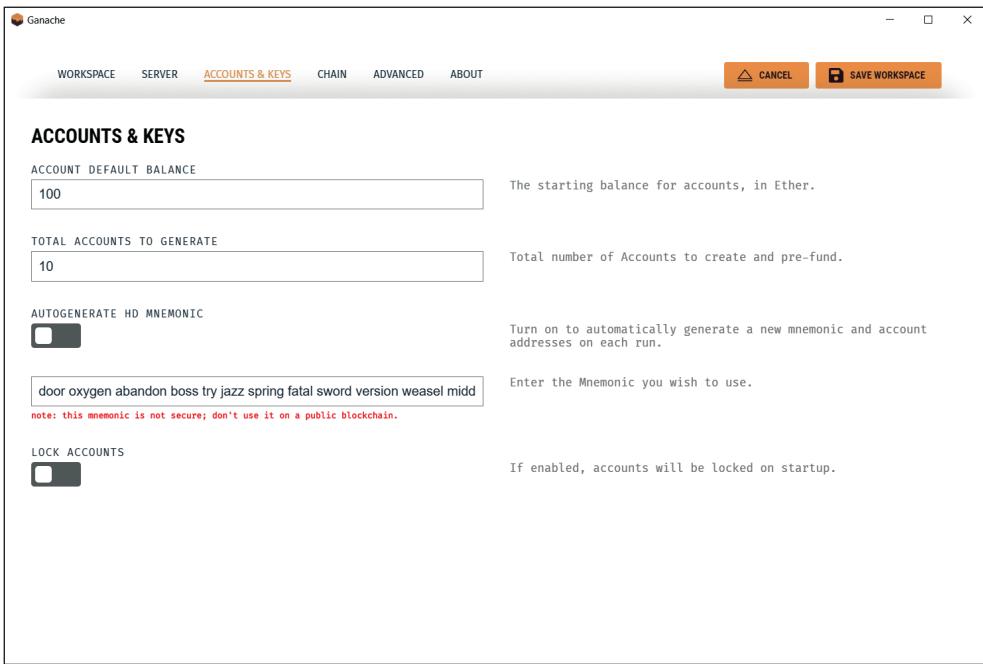


Рисунок 4.55. Вкладка «Accounts and Keys»

Вкладка «Chain» (рис. 4.56) используется для настройки самой цепочки. Здесь задаются параметры «Gas limit» и «Gas price» в соответствующих полях.

Если будет разработан слишком большой контракт, то для того, чтобы загрузить его в сеть, будет требоваться несколько больший лимит газа, поэтому можно увеличить значение параметра «Gas limit».

«Gas price» – это цена комиссии за переводы. Если поставить значение 0 в этом поле, то все действия в вашей сети будут бесплатными. Для первого тестового запуска эти два параметра можно оставить по умолчанию.

Выпадающее меню «Hardfork» задает настройки сети. В разделе 4.2.7 обсуждались версии системы «Константинополь» и «Византия», в которых есть предварительные настройки и дополнительные функции, отсутствующие в более ранних версиях системы. Если разработчику контракта необходимо посмотреть, как работают те или иные функции в новых форках, или посмотреть старые проекты, то существует возможность гибко подстроить систему под себя.

Вкладка «Advanced» содержит настройки приложения: работа с логами и Google-аналитика (все переключатели можно оставить выключенными).

Вкладка «About» содержит краткую информацию о приложении.

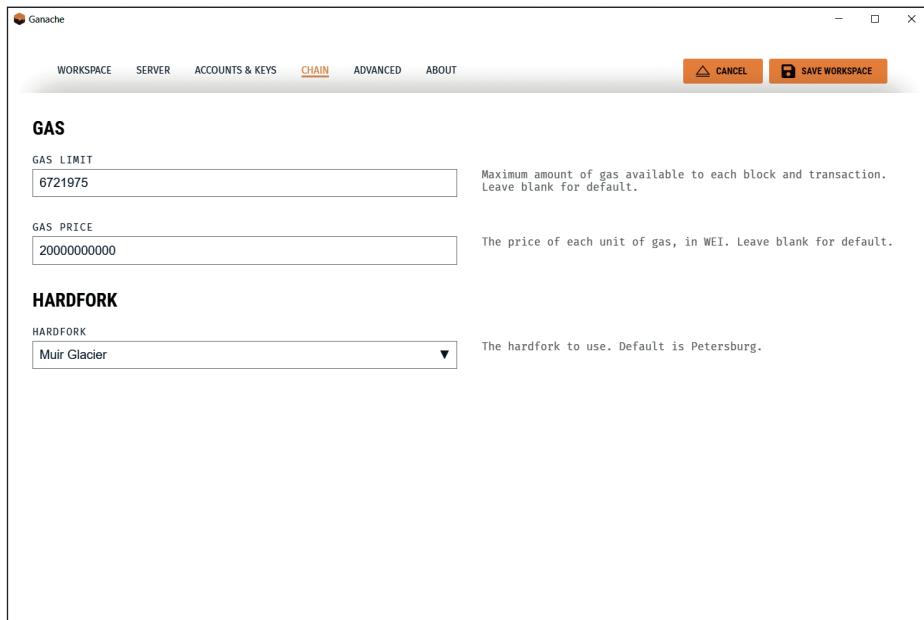


Рисунок 4.56. Вкладка «Chain»

После того как все параметры для настройки Ganache заданы, можно сохранить рабочее пространство и перейти на вкладку «Accounts». На вкладке откроется 10 адресов, баланс каждого адреса составит 100 Eth (рис. 4.57).

Ganache							SEARCH FOR BLOCK NUMBERS OR TX HASHES
ACCOUNTS	BLOCKS	TRANSACTIONS	CONTRACTS	EVENTS	LOGS	WORKSPACE HIGHLIGHTS	SWITCH
CURRENT BLOCK 0	GAS PRICE 2000000000	GAS LIMIT 6721975	HARDFORK MUIRGLEACIER	NETWORK ID 5777	RPC SERVER HTTP://127.0.0.1:7545	MINING STATUS AUTOMINING	
MNEMONIC <small>?</small> door oxygen abandon boss try jazz spring fatal sword version weasel middle							HD PATH m/44'/60'/0'/0/account_index
ADDRESS 0xeF7d36526a0db7aD2e38A3823040DeD005dd1671	BALANCE 100.00 ETH			TX COUNT 0	INDEX 0		
ADDRESS 0x9Cfd9D3019123b9eF3B6D2037FA0fA437615c852	BALANCE 100.00 ETH			TX COUNT 0	INDEX 1		
ADDRESS 0x3a4F1426328e7d5e35a3e194BF9c4D7c5ad91ea6	BALANCE 100.00 ETH			TX COUNT 0	INDEX 2		
ADDRESS 0x6cf8FdC7541982874C58b67AE26144d6b99eef42	BALANCE 100.00 ETH			TX COUNT 0	INDEX 3		
ADDRESS 0x62781D8ec57FA22e68FE314629e05BAD4e1bdEc8	BALANCE 100.00 ETH			TX COUNT 0	INDEX 4		
ADDRESS 0xB1dD6b32678A7D630309055EA4667E016b3AEb53	BALANCE 100.00 ETH			TX COUNT 0	INDEX 5		
ADDRESS 0x954ABb4cb6E7D59551Cd742c406a5d1Ba822f41c	BALANCE 100.00 ETH			TX COUNT 0	INDEX 6		

Рисунок 4.57. Созданные аккаунты тестовой сети

В небольшой панели под вкладками можно увидеть адрес сервера и номер сети, на которых запущен эмулятор, а также остальные поля, которые были заполнены при настройке приложения.

Поле «Current block» показывает номер текущего блока, но таких блоков еще нет в только что созданной цепочке. Есть единственный генезис-блок (начальный блок), высота которого равна 0.

Чтобы посмотреть информацию о блоке, можно просто нажать на него, после чего мы можем просмотреть его хеш-код, дату и время создания, лимит газа и сколько газа было использовано (рис. 4.58).

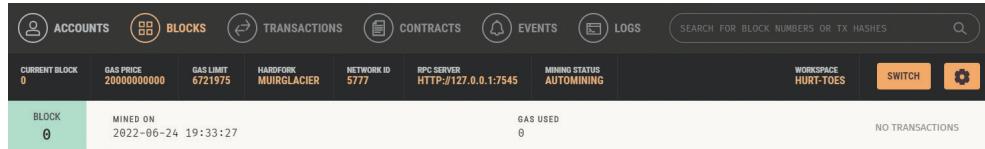


Рисунок 4.58. Информация о блоке

Обратите внимание на то, что майнинг в Эфириуме работает несколько иначе, чем в биткойне, и мы не видим нескольких нулей в начале хеш-кода просматриваемого блока. Здесь иные требования, которые основываются на вычислениях в графе.

Кратко опишем остальные элементы управления:

- вкладка «Transactions» содержит информацию о транзакциях. Пока их нет;
- параметр «Contracts» предназначен для загрузки контракта;
- вкладка «Events» отображает события;
- на вкладке «Accounts» есть также два следующих параметра: «Tx count» и «Index». Параметр «Tx count» показывает количество транзакций, созданных с данного адреса. Параметр «Index» показывает порядковый номер адреса в системе;
- параметр «Show keys» (символ ключика) содержит всю ключевую информацию: адрес пользователя и приватный ключ (рис. 4.59). Приватный ключ необходимо использовать только в целях разработки, никогда не используйте его в открытом виде в публичных блокчейнах.



Рисунок 4.59. Информация о закрытом ключе

Связь Ganache и MetaMask

Для того чтобы подключиться из приложения MetaMask к созданной эмулятором Ganache тестовой сети, необходимо в приложении MetaMask в списке сетей выбрать «Добавить сеть», ввести имя сети, добавить URL (адрес сервера: номер порта из приложения Ganache), идентификатор цепочки и обозначение криптовалюты «eth» (рис. 4.60).

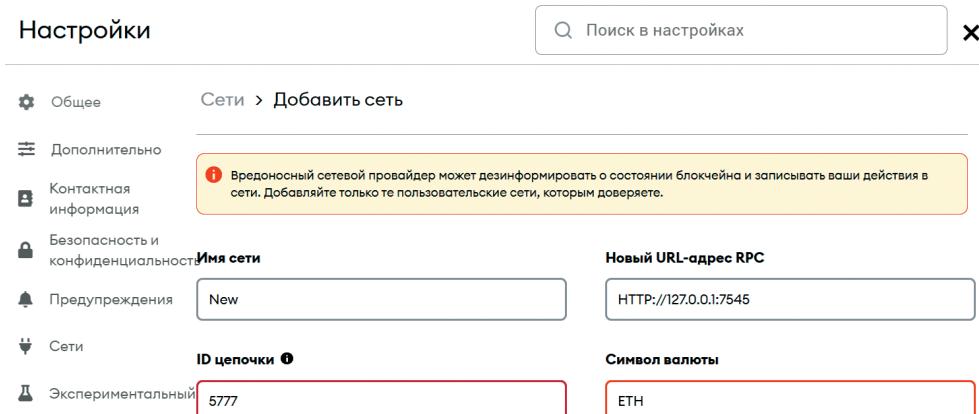


Рисунок 4.60. Загрузка в MetaMask данных сети Ganache

В списке «Мои счета» находятся кошельки из другой тестовой сети, нам необходимо загрузить сюда адреса из сети Ganache, передав кошельку закрытый ключ. Для этого на вкладке со счетами выбираем кнопку «Импортировать счет» и в появившемся окне вводим приватный ключ одного из адресов в Ganache.

Geth

Geth представляет собой полноценный консольный клиент сети Эфириум со всеми необходимыми настройками. Клиент Geth позволяет создавать частную сеть, а также работать с реальной сетью Эфириум [143].

Необходимо установить клиент, после чего можно начинать с ним работу. Работа происходит через ввод консольных команд в терминале.

Если в терминале запустить команду `geth`, то цепочка сети Эфириум будет скачиваться в папку «`C:\Пользователи\<имя пользователя>\AppData\Roaming\Ethereum\chaindata\<сеть>`».

Для создания новых пользователей в тестовой сети необходимо ввести следующую команду:

```
geth account new --datadir <путь к папке, в которую будем сохранять данные>
```

После ввода команды программа запрашивает ввод пароля. Этот пароль будет нужен в дальнейшем, поэтому обязательно нужно его сохранить. После ввода пароля на экран выводится адрес открытого ключа («Public address of the key») и путь к секретному ключу («Path of the secret key file») (рис. 4.61).

```

Administrator: C:\Windows\System32\cmd.exe
C:\Users\Станислав\Desktop\test>geth account new --datadir C:\test
INFO [01-17!17:37:25.163] Maximum peer count          ETH=50 LES=0
total=50
Your new account is locked with a password. Please give a password. Do not forget this password.
Password:
Repeat password:
Your new key was generated

Public address of the key: 0x9e0b0922914bc7955ee7bdaa284d3be06640350
Path of the secret key file: C:\test\keystore\UTC--2022-01-17T14-37-34.958512100Z--9ea0b0922914bc7955ee7bdaa284d3be06640350

- You can share your public address with anyone. Others need it to interact with you.
- You must NEVER share the secret key with anyone! The key controls access to your funds!
- You must BACKUP your key file! Without the key, it's impossible to access account funds!
- You must REMEMBER your password! Without the password, it's impossible to decrypt the key!

C:\Users\Станислав\Desktop\test>

```

Рисунок 4.61. Создание нового аккаунта с помощью Geth

Если перейти по указанному адресу (зайти в только что созданный кошелек), то можно увидеть данные кошелька, которые записаны в JSON-формате (рис. 4.62).

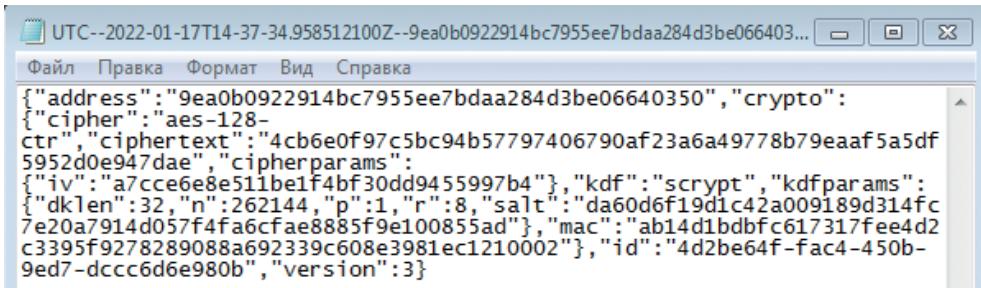


Рисунок 4.62. Файл кошелька

Здесь указаны адрес и параметры криптографической системы, которые показывают нам способ шифрования, параметры используемого алгоритма, а также ряд других данных.

Использование шифрования и сохранение данных в зашифрованном виде – это блокировка адреса или кошелька в системе, которая позволяет защитить пользователя от исполнения команд посторонним лицом. Пока пользователь не введет пароль, тем самым подтверждая, что он может расшифровать сообщение и получить закрытый ключ, он не сможет использовать данный ключ для подписи транзакций.

Создадим файл генезис-блока, указав там необходимые настройки. Например, создадим при старте системы два адреса с разными балансами 1000 и 10 Eth (рис. 4.63).

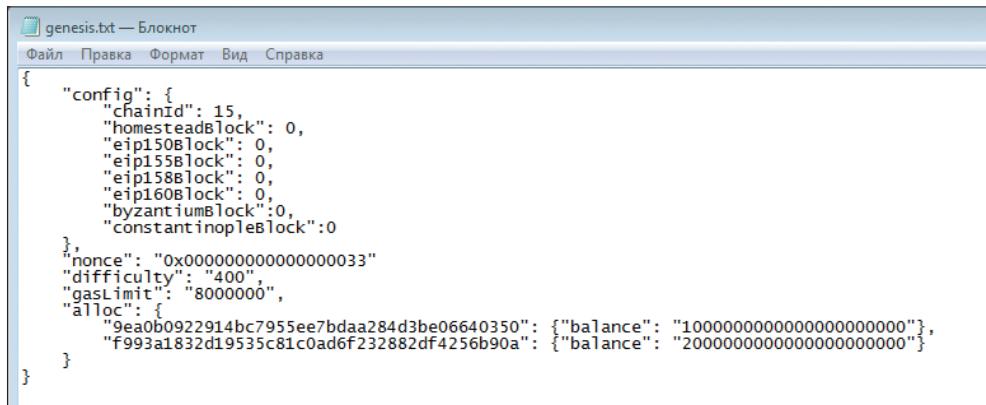


Рисунок 4.63. Файл генезис-блока

Для этого перейдем в терминал и введем следующую команду (рис. 4.64):

```
geth --datadir <директория для записи цепочки> init <путь к генезис-блоку>
```

После успешного выполнения появится сообщение о том, что состояние генезиса успешно записано.

```
C:\Users\Станислав\Desktop\test>geth --datadir C:\test init C:\test\genesis.json
INFO [01-17|17:57:43.078] Maximum peer count                                     ETH=50 LES=0
total=50
INFO [01-17|17:57:43.473] Set global gas cap                                         cap=250000000
INFO [01-17|17:57:43.476] Allocated cache and file handles                         database=C:\t
est\geth\chaindata cache=16.00MiB handles=16
INFO [01-17|17:57:43.486] Writing custom genesis block
INFO [01-17|17:57:43.488] Persisted trie from memory database                      nodes=3 size=
409.00B time=0s gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [01-17|17:57:43.493] Successfully wrote genesis state                         database=chai
nmeta hash="41360e:d8d83d"
INFO [01-17|17:57:43.496] Allocated cache and file handles                         database=C:\t
est\geth\lightchaindata cache=16.00MiB handles=16
INFO [01-17|17:57:43.504] Writing custom genesis block
INFO [01-17|17:57:43.506] Persisted trie from memory database                      nodes=3 size=
409.00B time=0s gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [01-17|17:57:43.511] Successfully wrote genesis state                         database=ligh
tchaindata hash="41360e:d8d83d"

C:\Users\Станислав\Desktop\test>
```

Рисунок 4.64. Инициализация генезис-блока

При этом появится папка «geth», в которой будет накапливаться цепочка. Также будут созданы две папки: «chaindata» и «lightchaindata».

Чем они отличаются? Если вы хотите стать легким клиентом цепи, то вам достаточно подгрузить из интернета «lightchaindata» – это просто хеш-коды всех блоков, облегченная версия цепочки, которая весит гораздо меньше основной, что позволяет работать с цепочкой, но майнить блоки при этом нельзя.

Если же вы хотите полноценно работать с сетью, то для этого нужна папка «chaindata». При инициализации в цепочке существует только нулевой блок, который не содержит транзакций, поэтому эти две папки идентичны.

Теперь необходимо произвести инициализацию цепочки. Для этого необходимо в консоли ввести команду следующего формата:

```
geth --grpc --grpcapi "eth, miner, personal, txpool, admin, web3" --datadir <путь к нашей цепочке> --rpccaddr "адрес" --networkid 15 --allow-insecure-unlock --rpccorsdomain "*" console
```

В нашем случае команда запуска выглядит так (рис. 4.65):

```
geth --grpc --grpcapi "eth, miner, personal, txpool, admin, web3" --datadir C:\test --rpccaddr "127.0.0.1" --networkid 15 --allow-insecure-unlock --rpccorsdomain "*" console
```

```
C:\Users\Станислав\Desktop\test>geth --rpc --rpccapi "eth, miner, personal, txpool, web3" --datadir C:\test --rpccaddr "127.0.0.1:8545" --networkid 15 --allow-insecure-unlock --rpccorsdomain "*" console
WARN [01-17!18:07:58.608] Lowering memory allowance on 32bit arch available=16235 addressable=2048
WARN [01-17!18:07:58.613] Sanitizing cache to Go's GC limits provided=1024 updated=682
INFO [01-17!18:07:58.616] Maximum peer count total=50 ETH=50 LES=0
WARN [01-17!18:07:58.619] The flag --rpc is deprecated and will be removed in the future, please use --http
WARN [01-17!18:07:58.622] The flag --rpccaddr is deprecated and will be removed in the future, please use --http.addr
```

Рисунок 4.65. Инициализация цепочки

Рассмотрим, что означают введенные команды и параметры:

- --rpc – протокол удаленного вызова команд;
- --grpcapi – указываем используемые модули geth;
- eth – отвечает за работу цепочки;
- miner – отвечает за работу с майнингом;
- personal – работа с определенным кошельком;
- txpool – отвечает за транзакции;
- web3 – предоставляет интерфейс разработки приложений (Application Programming Interface, API) к нашей цепочке;
- --datadir – указываем, с какой цепочкой будем работать;
- --rpccaddr – адрес, с которым будет работать протокол (указываем адрес из Ganache);
- --networkid 15 – идентификатор сети;
- --allow-insecure-unlock – упрощает работу с ключами;
- --rpccorsdomain – позволяет подключаться разным сервисам к нашей цепочке;
- console – запускает сервис в режиме консоли.

После запуска можем работать с консолью Geth, которая поддерживает синтаксис языка JavaScript. Можно пользоваться модулями, которые были подключены в --grpcapi.

Первый модуль – модуль eth. Для того чтобы посмотреть, какие команды для него доступны, нужно в консоль ввести команду eth. (с точкой в конце) и быстро нажать два раза клавишу Tab, после чего откроется список доступных команд (рис. 4.66).

```

> eth.
eth._requestManager
eth.accounts
eth.blockNumber
eth.call
eth.chainId
eth.coinbase
eth.compile
eth.constructor
eth.contract
eth.defaultAccount
eth.defaultBlock
eth.estimateGas
eth.fillTransaction
eth.filter
eth.gasPrice
eth.getAccounts
eth.getBalance
eth.getBlock
eth.getBlockByHash
eth.getBlockByNumber
eth.getBlockNumber
eth.getBlockTransactionCount
eth.getBlockUncleCount
eth.getCode
eth.getCoinbase
eth.getCompilers
eth.getGasPrice
eth.getHashrate
eth.getHeaderByHash
eth.getHeaderByNumber
eth.getMining
> eth.INFO [01-17:18:18:26.393] Looking for peers
nt=1 tried=97 static=0

```

Рисунок 4.66. Команды модуля eth

Названия команд интуитивно понятны. Так, например:

- eth.blockNumber – номер текущего блока в цепочке;
- eth.accounts – список аккаунтов, которые есть в сети;
- eth.getBalance – запрос баланса какого-либо пользователя;
- eth.getBlock – получить информацию о блоке;
- eth.getHashrate – получить хешрейт;
- eth.getUncle – можно узнать дядей блока;
- eth.sendTransaction – отправить транзакцию;
- eth.sign – подписать транзакцию и т. д.

Для того чтобы посмотреть, какие аккаунты сейчас есть в сети, необходимо ввести команду eth.accounts. После выполнения команды на экран будут выведены те адреса, которые находятся в рабочей папке (те же адреса, что и в генезис-блоке) (рис. 4.67). Если генезис-блок был сформирован с ошибкой, но инициализировал цепочку, то у этих адресов будет нулевой баланс.

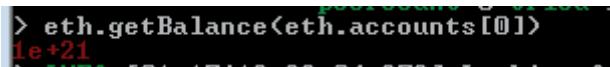
```

> eth.accounts
["0x9ea0b0922914bc7955ee7bdaa284d3be06640350", "0xf993a1832d19535c81c0ad6f232882
df4256b9@a"]

```

Рисунок 4.67. Просмотр аккаунтов в сети

Баланс двух аккаунтов на рис. 4.67 обуславливается значением, заданным в генезисе. Для просмотра баланса первого по порядку (с индексом 0) аккаунта необходимо ввести команду `eth.getBalance(eth.accounts[0])` (так как `accounts` – это список с адресами, то баланс адреса можно узнать, обратившись к нему по его индексу). В консоли появится баланс выбранного аккаунта (рис. 4.68).



```
> eth.getBalance(eth.accounts[0])
1e+21
```

Рисунок 4.68. Просмотр баланса аккаунта по индексу

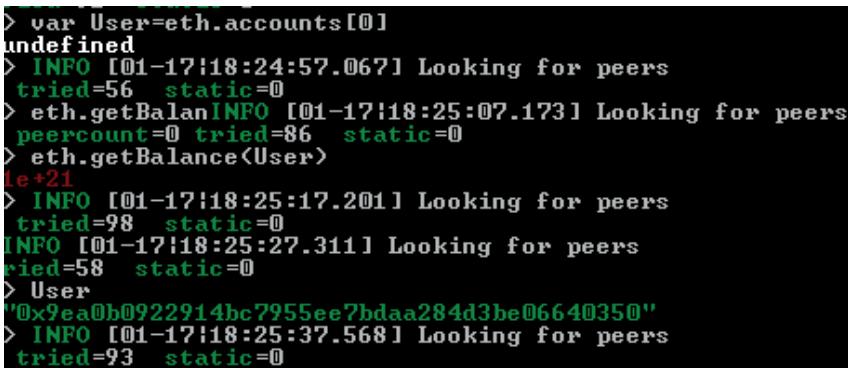
Второй способ посмотреть баланс – ввести команду `eth.getBalance(«адрес»)`. Есть также и третий способ, основанный на использовании JavaScript, – для просмотра баланса следует создать какую-нибудь переменную и присвоить ей значение, например `eth.accounts[0]` (рис. 4.69):

```
var User = eth.accounts[0],
```

После этого передаем созданную переменную в команду:

```
eth.getBalance(User)
```

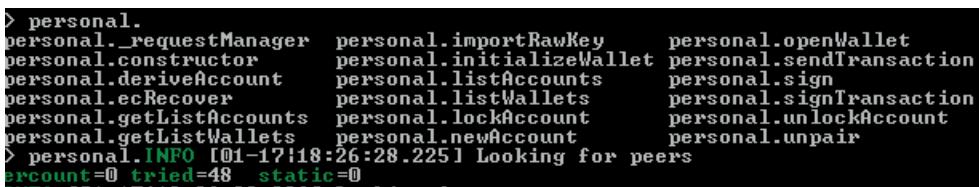
Если теперь в консоли просто ввести имя переменной `User`, то на экран будет выведено значение адреса, которое в ней хранится.



```
> var User=eth.accounts[0]
undefined
> INFO [01-17:18:24:57.067] Looking for peers
  tried=56 static=0
> eth.getBalancINFO [01-17:18:25:07.173] Looking for peers
  peercount=0 tried=86 static=0
> eth.getBalance(User)
1e+21
> INFO [01-17:18:25:17.201] Looking for peers
  tried=98 static=0
INFO [01-17:18:25:27.311] Looking for peers
  tried=58 static=0
> User
"0x9ea0b0922914bc7955ee7bdaa284d3be06640350"
> INFO [01-17:18:25:37.568] Looking for peers
  tried=93 static=0
```

Рисунок 4.69. Просмотр баланса аккаунта с помощью дополнительной переменной

Для просмотра команд модуля `personal` необходимо аналогичным образом ввести в консоль его название (с точкой в конце) и два раза нажать **Tab**. Как и в случае с `eth`, на экран будет выведен полный список команд (рис. 4.70).



```
> personal.
personal._requestManager  personal.importRawKey    personal.openWallet
personal.constructor      personal.initializeWallet personal.sendTransaction
personal.deriveAccount    personal.listAccounts   personal.sign
personal.ecRecover        personal.listWallets   personal.signTransaction
personal.getListAccounts  personal.lockAccount  personal.unlockAccount
personal.getListWallets   personal.newAccount   personal.unpair
> personal.INFO [01-17:18:26:28.225] Looking for peers
  peercount=0 tried=48 static=0
```

Рисунок 4.70. Команды модуля `personal`

Функция personal.newAccount создает новый аккаунт (адрес). Так как это функция, то в консоли при вводе данной команды нужно указать скобки: personal.newAccount(). Если скобки оставить пустыми, то после выполнения данной команды последует просьба придумать пароль, после ввода которого будет выдан новый адрес.

Кроме того, можно сразу в функцию передать пароль, указав его в скобках (обязательно в кавычках), например так (рис. 4.71):

```
personal.newAccount("123")
> personal.newAccount("123")
INFO [01-17:18:27:42.924] Your new key was generated address=0xF25
56f2d0839c556317a9d403349cc58dab2e29d
WARN [01-17:18:27:42.929] Please backup your key file! path=C:\test\
keystore\UTC--2022-01-17T15-27-40.855439600Z--f2556f2d0839c556317a9d403349cc58da
b2e29d
WARN [01-17:18:27:42.937] Please remember your password!
"0xf2556f2d0839c556317a9d403349cc58dab2e29d"
> INFO [01-17:18:22:48.659] Looking for peers peercount=0
```

Рисунок 4.71. Создание нового аккаунта

Функция personal.unlockAccount используется для разблокировки аккаунта. Модуль miner содержит две важные функции:

- miner.start() – функция, запускающая майнинг. После выполнения консоль обновляется, начинают появляться сообщения со статусом состояния блока. Если во время выполнения мы захотим узнать баланс пользователя, то после выполнения команды увидим, что баланс майнера постоянно увеличивается;
- miner.stop() – функция, останавливающая процесс майнинга.

4.2.10 Смарт-контракты в системе Эфириум

Понятие смарт-контрактов было описано ранее в разделе 3.4. Рассмотрим далее специфику применения смарт-контрактов на платформе Эфириум.

Отметим, что смарт-контракты в системе Эфириум предоставляют значительно более широкие возможности по сравнению с рассмотренными ранее смарт-контрактами в системе Биткойн.

Токены

На сегодняшний день существует несколько специализированных шаблонов смарт-контрактов – токенов. Под токеном понимается цифровой актив, который принадлежит пользователю системы. Смарт-контракт содержит в себе необходимый набор функционала по работе с токеном.

Фактически под владением токеном понимается привязка определенного значения числовой переменной к адресу пользователя. Стоимость такого цифрового актива будет варьироваться в зависимости от спроса на него и от того, как владельцы каждого токена будут ценить владение им. Наиболее популярные токены соответствуют стандарту ERC-20 [116], который будет подробно описан далее.

Использование шаблонного кода смарт-контракта позволяет отслеживать токен различным сервисам, как криптокошелькам, так и биржам, обменникам.

Один из наиболее популярных обменников – сервис Uniswap [238]. Сервис позволяет обменивать токены на фиатную валюту, криптовалюту или на другие токены, автоматически рассчитывая стоимость обмена.

Описание токена ERC-20

Токен ERC-20 содержит в своем составе следующий набор функций:

- name() – имя токена, заданное владельцем токена при его выпуске;
- symbol() – символическое обозначение токена, заданное владельцем токена при его выпуске; в обозначении используются заглавные латинские символы, например SHIB, WETH, DAI;
- decimals() – количество знаков после запятой, позволяет разделить токен на дробные части; например, если значение равно трем, то каждый токен может быть разделен на 1000 составных частей. Обычно используется значение decimal = 18, по аналогии с соотношением Eth и wei;
- totalSupply() – общее количество токенов в системе;
- balanceOf(address _owner) – баланс токенов конкретного пользователя;
- transfer(address _to, uint256 _value) – перевод токенов с баланса одного пользователя другому;
- approve(address _spender, uint256 _value) – разрешить другому пользователю тратить свои токены в пределах указанного количества;
- allowance(address _owner, address _spender) – количество токенов, которые пользователь разрешил тратить другому со своего баланса;
- transferFrom(address _from, address _to, uint256 _value) – перевести токены пользователя, доступ к которым был предоставлен, другому пользователю.

Обычно перед разработчиками становится задача увеличить функционал токена, добавить дополнительную бизнес-логику. Рассмотрим язык программирования Solidity, который обычно применяется для разработки смарт-контрактов.

Язык программирования Solidity

Язык Solidity был разработан в 2014 году. Язык является кросс-платформенным, однако применяется преимущественно для смарт-контрактов платформы Эфириум.

Solidity представляет собой объектно-ориентированный, статически типизованный JavaScript-подобный компилируемый язык программирования. Спустя годы использования язык не имеет завершенной версии, и разработчики постоянно выпускают новые релизы, совершенствуя и развивая функционал [227].

Синтаксис и компиляцию кода поддерживают многие среды разработки и редакторы кода, в их числе VisualStudio Code, Sublime Text, Atom, Remix IDE, IntelliJ IDEA и др.

Структура кода

Рассмотрим основную структуру кода на языке Solidity.

В первой строке всегда указывается версия используемого компилятора:

```
pragma solidity (>=)0.7.0
```

Дальше создадим класс контракта – например, `NewContract`. Вместо ключевого слова `class`, как в других языках программирования, указывается слово `contract`, после имени класса в фигурных скобках будет находиться тело контракта:

```
contract contractName {}.
```

В начале описания контракта можно объявить глобальные переменные.

После переменных создается конструктор – функция, которая выполняется при запуске, публикации или размещении контракта, в фигурных скобках приводится тело конструктора:

```
constructor {}
```

После конструктора можно создавать свои функции. Функция объявляется с помощью ключевого слова `function`, после которого указывается имя функции, при этом в последующих фигурных скобках приводится само тело функции.

Пример основной структуры контракта приведен на рис. 4.72.

Структура кода:

<code>pragma solidity (>=)0.7.0;</code>	указываем версию языка
<code>contract NewContract {</code>	объявляем контракт
<code>uint a;</code>	глобальные переменные
<code>bool b;</code>	
<code>address Vova = 0xAc771378BB6c2b8878fbF75F80880cbdDefd1B1e;</code>	
<code>constructor {</code>	конструктор – функция, которая выполняется при
<code>.....</code>	публикации/запуске/деплое контракта
<code>}</code>	
<code>function MyFunction {</code>	функция – вызываются при определенных событиях
<code>.....</code>	
<code>}</code>	

Рисунок 4.72. Основная структура контракта на языке Solidity

Использование памяти и типизация переменных

Существует два типа памяти для переменных в контракте:

- глобальная – `storage` – для хранения глобальных переменных (см. пример на рис. 4.72);
- локальная – `memory` – для переменных, которые используются в функциях.

При этом в языке Solidity использование глобальных переменных несет в себе дополнительные издержки, потому что состояние глобальной переменной записывается в глобальное состояние контракта, глобальное состояние всей системы и запись в эту область памяти стоят гораздо дороже для исполнителя этой функции (речь идет о количестве газа, который потребуется затратить для выполнения контракта).

В Solidity существуют следующие типы переменных:

- bool – логический тип;
- uint – беззнаковые целые числа; причем для каждой переменной этого типа можно явно указать ее размер в битах, кратный 8: например, uint8, uint32, uint128...; по умолчанию размер равен 256, т. е. объявление uint эквивалентно uint256;
- bytes – байтовые последовательности, размер которых задается аналогично типу uint, но не более 32;
- string – строки;
- array[] – массивы; для чисел с типом uint массив будет объявляться как uint[], для строк – string[] и т. д.;
- struct – различные пользовательские структуры (см. пример далее);
- mapping (address =>) balances – словари, которые по ключу хранят значение;
- address – адрес пользователя или контракта, который представляет собой последовательность из 40 символов (20 байт).

Пользовательские структуры объявляются аналогично многим другим языкам программирования, например:

```
struct{
    uint a;
    uint b;
}
```

Поговорим о структурах немного подробнее. Рассмотрим пример исходных данных системы, которые заданы в соответствии с табл. 4.13.

Таблица 4.13. Пример задания исходных данных системы

estate_id	owner	info	square	useful_square	present_status	sale_status
(целое число)	(адрес Eth)	(строка)	(целое число)	(целое число)	(True/False)	(True/False)
0	0xfCC..90C	Чехова 2	150	130	False	False
1	0x148..44B	Фрунзе 9	70	60	True	False

Для хранения данных, соответствующих табл. 4.13, можно составить следующую структуру:

```
struct Estate {
    uint estate_id;
    address owner;
    string info;
    uint square;
    uint useful_square;
    bool present_status;
    bool sale_status;
}
```

То есть для каждого поля в первой строке таблицы указывается тип данных, который указан во второй строке таблицы.

Обращение к элементам структуры происходит следующим образом (рис. 4.73):

- если необходимо получить всю структуру, то просто указывается ее название;
- если необходимо получить конкретный элемент, то нужно указать его в следующем формате: название_структурь[искомый элемент];
- если необходимо получить конкретный параметр элемента, то нужно обратиться к нему в формате: название_структурь[искомый элемент].параметр.

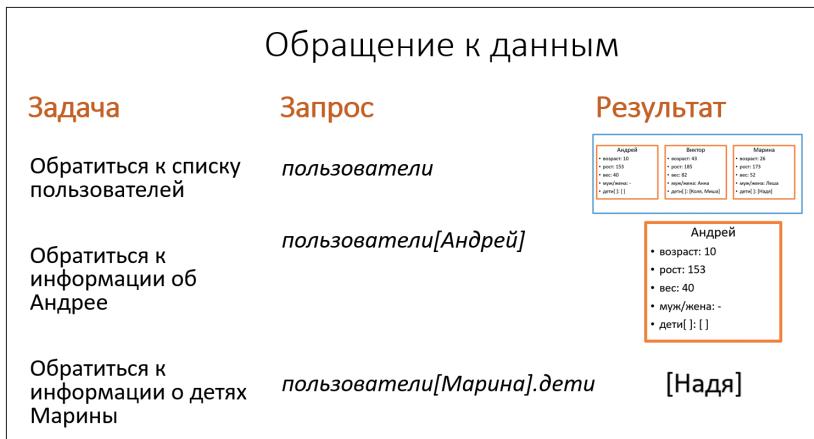


Рисунок 4.73. Обращение к элементам структуры

Функции

Функции объявляются с использованием ключевого слова `function`, после которого указывается имя функции, а также параметры (типы данных и имена переменных, которые принимает функция) и атрибуты функции. Если функция будет возвращать какие-то значения, то типы данных этих значений необходимо указать в списке возвращаемых значений `returns` (см. рис. 4.74).

Структура функции

```
function my_function(type var, ....)
    атрибуты функции
    тело функции
}
```

атрибуты функций

`public` – вызывается из любого контракта

Модификаторы доступа
(`global / local`)

`private` – вызывается только в пределах этого контракта

`view` – **чтение**, функция только возвращает данные из памяти, не изменяя их

`payable` – в ходе выполнения функции изменяются балансы

Рисунок 4.74. Структура функции

Затем в фигурных скобках создается тело функции. Если функция возвращает значения, то их необходимо в теле функции вернуть с помощью команды `return <имя переменной>`. При этом важно соблюдать порядок возвратов с порядком типов, указанных в блоке `returns`.

Атрибуты функций приведены на рис. 4.74. Атрибут `public` позволяет вызвать функцию одного контракта из любого участка кода другого контракта, а атрибут `private` разрешает вызов функции только в том контракте, в котором функция находится.

Основные операторы

В языке Solidity есть сложные операторы, такие как `if-else`, циклы `for`, `while`, конструкция `require`. Однако цикл `while` использовать не рекомендуется.

Конструкция `require` используется следующим образом: если условие выполняется, то осуществляется переход к следующей команде в теле этой конструкции; если условие не выполняется, то выполнение функции прекращается, выводится сообщение об ошибке, транзакция не формируется.

В ходе выполнения функции можно обращаться к параметрам транзакции. Транзакция обозначается как `msg`, имеет два основных атрибута `msg.sender` (адрес создателя транзакции) и `msg.value` (сумма транзакции в wei).

Если необходимо работать с криптовалютой, то при перечислении средств с обычного аккаунта на кошелек смарт-контракта деньги перечисляются автоматически с заполнением поля `msg.value` при формировании транзакции.

Если криптовалюту нужно перечислить с кошелька контракта на кошелек пользователя или кошелек другого контракта, то необходимо использовать следующую конструкцию (при этом сумма перевода указывается в wei):

```
<address>.transfer(<amount>)
```

Также очень полезной функцией является `block.timestamp(now)`. Это метка времени в формате Unix time в момент создания блока с транзакцией (по времени блокчейн-системы). Значение имеет тип `uint`.

Среда разработки Remix IDE

Наиболее популярным средством для разработки смарт-контракта является Remix IDE [207] – онлайн-среда разработки.

Среда разработки содержит тестовые контракты (слева на вкладке `contracts`). Рассмотрим один из тестовых контрактов (рис. 4.75).

Представленный на рисунке контракт содержит переменную `number` и два метода. При этом функция `store` задает значение поля `number`, а функция `retrieve` возвращает данное значение.

Среда разработки включает в себя различные виртуальные машины. Так, например, виртуальная машина Injected Web 3 может работать с расширением MetaMask, которое было рассмотрено выше.

```

1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6  * @title Storage
7  * @dev Store & retrieve value in a variable
8  */
9 contract Storage {
10
11     uint256 number;
12
13     /**
14      * @dev Store value in variable
15      * @param num value to store
16      */
17     function store(uint256 num) public {
18         number = num;
19     }
20
21     /**
22      * @dev Return value
23      * @return value of 'number'
24      */
25     function retrieve() public view returns (uint256){
26         return number;
27     }
28 }
```

Рисунок 4.75. Тестовый контракт

Выбрать необходимую машину можно в разделе environment, после чего автоматически произойдет сопряжение с расширением MetaMask и будет предложено выбрать и импортировать счет для работы (рис. 4.76).

Для того чтобы контракт оказался размещенным в сети Эфириум, необходимо выполнить функцию деплоя (deploy). После компиляции кода контракта на специальной вкладке необходимо нажать кнопку «Deploy». При этом откроется вкладка MetaMask с подтверждением операции. После успешного выполнения в терминале среды разработки появляется зеленая галочка и сообщение о том, что все выполнено успешно. Также мы сможем увидеть контракт на вкладке Deployed contracts.

Есть еще один вариант подключения к цепочке – это использование Web 3 Provider, который связан с запущенным клиентом Geth. При нажатии на Web 3 Provider появляется всплывающее окно, в котором нам необходимо указать адрес и порт.

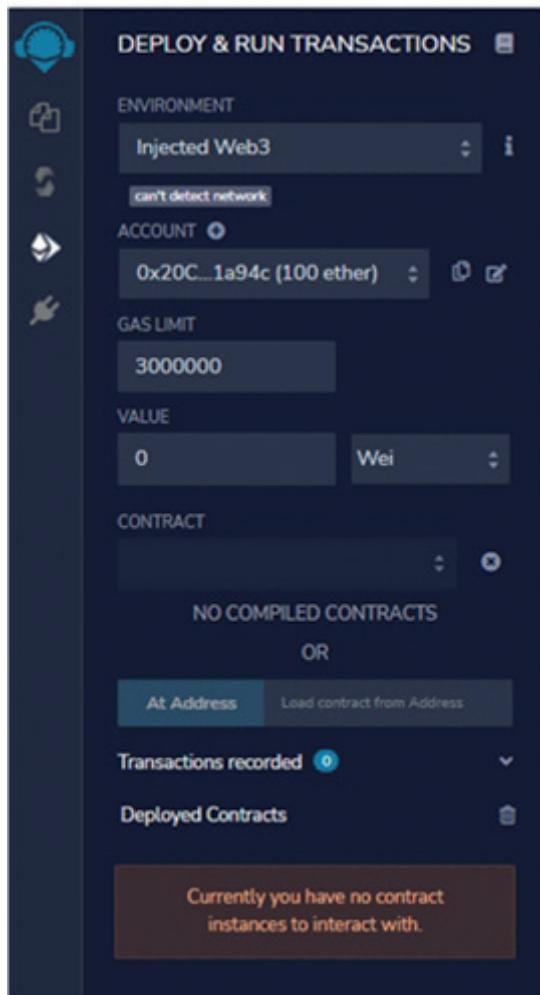


Рисунок 4.76. Выбор виртуальной машины

При попытке деплоить контракт может возникнуть ошибка, связанная с отсутствием доступа. Для решения этой проблемы необходимо в консоли клиента Geth разблокировать аккаунт следующим образом:

```
personal.unlockAccount(<указываем, какой аккаунт хотим разблокировать>,
<пароль>, <время, на которое разблокируем пользователя, - необязательный
параметр>)
```

Далее необходимо запустить майнинг командой `miner.start()`. После этого можно вернуться в среду разработки и выполнить деплой контракта.

Пример разработки смарт-контракта

Покажем, как разработать смарт-контракт, на примере. Пусть перед нами стоит задача разработать систему перевода внутрисистемных средств с пароль-

ной защитой перевода – пользователь должен ввести корректный код для завершения перевода и получения средств.

В первую очередь указывается версия используемого компилятора, допускаются операторы сравнения. Стоит отметить, что не все команды программного кода будут обратно совместимы с устаревшими версиями компилятора.

```
pragma solidity ^0.5.0;
```

Как было сказано выше, контракт фактически представляет собой класс – имеет атрибуты, конструктор и методы.

```
contract Change {
```

```
...
```

Как и другие языки, Solidity поддерживает пользовательский тип данных в виде структур. Объявим структуру с балансом пользователя и его ролью в системе.

```
struct User {
    uint balance;
    bool role;
    bool registered;
}
```

Доступ к структуре может быть получен с помощью соответствия (маппинга, словаря), которое по уникальному значению адреса пользователя будет хранить информацию о нем. Атрибут public обеспечит получение доступа к этим данным.

```
mapping(address => User) public users;
```

Маппинг не позволяет получить список заполненных уникальных ключей (в данном случае – адресов), по любому ключу будет возвращена информация, даже пустая. Для решения этой проблемы создадим список используемых ключей.

```
address[] public userlist;
```

Опишем структуру перевода: будем фиксировать адрес отправителя средств, адрес получателя и объем перевода. Помимо этого, добавим в систему возможность указания категории перевода. Для обеспечения защиты перевода кодовым словом введем переменную для хранения хеш-кода пароля. Предусмотрим также переменную для хранения времени успешного завершения транзакции и логическую переменную для фиксации завершения транзакции.

Эти два поля будут характеризовать перевод:

- если оба значения нулевые, то перевод сформирован;
- если время перевода равно нулю, но перевод завершен, то перевод был отменен отправителем или получатель ввел неверный код;
- если время перевода указано и перевод завершен, то код был верно указан и получатель получил предназначенные ему средства.

```
struct Transfer {
    address adr_from;
    address adr_to;
    uint value;
    uint category;
    bytes32 pwhash;
    uint time;
    bool finished;
}
```

Заведем в системе массив переводов, каждый из которых будет доступен по его индексу.

```
Transfer[] public transfers;
```

Также заведем список категорий для переводов.

```
string[] public categories;
```

Как и класс, смарт-контракт имеет конструктор или функцию инициализации – функцию, которая будет выполнена сразу при публикации контракта в сети.

Зафиксируем в системе одного пользователя, администратора системы, обозначим его стартовый баланс в 10 000 условных единиц. Добавим в список пользователей структуру с нужными параметрами и добавим этот адрес к списку пользователей.

Конструктор выполняется от имени пользователя, который деплоит контракт, он будет распознан как msg.sender в этой функции. Также добавим несколько типов переводов в соответствующий список.

```
constructor() public {
    users[msg.sender] = User(10000, true, true);
    userlist.push(msg.sender);
    categories.push("Личный перевод");
    categories.push("Аренда жилья");
}
```

Создадим функцию регистрации нового пользователя. Если в значении флага registered у пользователя уже установлено значение true, то выполнение транзакции будет прервано. Если пользователь еще не был зарегистрирован, то заполнятся необходимые структуры.

```
function create_user() public {
    require(users[msg.sender].registered == false);
    users[msg.sender] = User(0, false, true);
    userlist.push(msg.sender);
}
```

Для реализации ролевой модели в системе добавим модификатор контроля прав администратора: если роль вызывающего функцию пользователя – не администратор, то модификатор прервет дальнейшее выполнение функции.

```
modifier onlyAdmin() {
    require(users[msg.sender].role, "errorg: you are not admin");
    _;
}
```

Добавим функцию создания новой категории. С помощью модификатора доступ к данной функции предоставлен только администратору. Новая категория добавляется в соответствующий список.

```
function create_category(string memory name) public onlyAdmin {
    categories.push(name);
}
```

Определим функцию создания перевода. Функция проверяет объем средств у пользователя, проверяет наличие в системе указанной категории, проверяет существование в системе получателя перевода. Функция принимает в качестве параметров адрес получателя, сумму перевода, категорию перевода и хеш-код кодового слова.

Хеш-код может быть получен в клиентском приложении или высчитан пользователем самостоятельно в случае обращения непосредственно к контракту. Используемый алгоритм хеширования – Keccak (см. раздел 1.3).

В случае успешного прохождения всех проверок будет создана новая структура с переводом, а баланс отправителя уменьшится на соответствующее значение. При создании перевода время выполнения будет равно нулю, флаг перевода будет равен false.

```
function create_transfer(address adr_to, uint value, uint category, bytes32
pwhash) public {
    require(users[msg.sender].balance >= value, "errorg: not enough money");
    require(users[adr_to].registered == true, "errorg: recipient is not
registered");
    require(category>=0 && category<categories.length, "errorg: wrong category");
    transfers.push(Transfer(msg.sender, adr_to, value, category, pwhash, 0, false));
    users[msg.sender].balance -= value;
}
```

Введем функцию отмены перевода отправителем. Если перевод относится к этому пользователю и не был завершен ранее, то флаг завершения установится в true и средства вернутся на счет отправителя.

```
function stop_my_transfer(uint tr_id) public {
    require(transfers[tr_id].adr_from == msg.sender, "errorg: not your transfer");
    require(!transfers[tr_id].finished, "error: already finished");
    transfers[tr_id].finished = true;
    users[msg.sender].balance += transfers[tr_id].value;
}
```

Опишем также функцию ввода кодового слова и завершения перевода. В случае если перевод относится к пользователю, перевод еще не был завершен и получатель ввел верный код, то будет зафиксировано время перевода, статус завершения установится в true, а баланс получателя увеличится.

Если код был введен неверно, то флаг завершения также устанавливается в true, а средства возвращаются на счет отправителя. Код может быть введен в строковом формате, хеш-код будет вычислен самим контрактом.

```
function get_my_transfer(uint tr_id, string memory pw) public {
    require(transfers[tr_id].adr_to == msg.sender, "error: transfer is not for
you");
    require(!transfers[tr_id].finished, "error: already finished");
```

```

bytes32 pwhash = keccak256(abi.encodePacked(pw));
if (pwhash == transfers[tr_id].pwhash) {
    transfers[tr_id].time = block.timestamp;
    users[msg.sender].balance += transfers[tr_id].value;
}
else {
    address adr_from = transfers[tr_id].adr_from;
    users[adr_from].balance += transfers[tr_id].value;
}
transfers[tr_id].finished = true;
}

```

На этом разработку смарт-контракта можно считать завершенной.

4.3 HYPERLEDGER

4.3.1 Основные особенности системы

Требования к системам на основе блокчейн-сети в зависимости от сферы бизнеса различаются. Для некоторых сфер применения требуются системы на основе быстрого сетевого консенсуса и короткое время подтверждения перед добавлением блока в цепочку. Для других более медленное время обработки может быть приемлемо в обмен на более низкие уровни требуемого доверия.

Масштабируемость, конфиденциальность, соответствие требованиям, сложность рабочего процесса и даже требования к безопасности сильно различаются в зависимости от отрасли и сферы использования. Каждое из этих требований представляет собой потенциально уникальную точку оптимизации для технологии.

По этим причинам Hyperledger инкубирует и продвигает ряд технологий бизнес-блокчейнов, включая распределенные реестры, механизмы смарт-контрактов, клиентские библиотеки, графические интерфейсы, служебные библиотеки и примеры приложений. Стратегия «зонтика» Hyperledger поощряет повторное использование общих строительных блоков через модульную архитектурную структуру.

Это позволяет быстро внедрять инновации в технологии распределенного реестра (Distributed Ledger Technologies, DLT), общие функциональные модули и интерфейсы между ними. Преимущества этого модульного подхода включают расширяемость, гибкость и возможность независимо изменять любой компонент, не затрагивая остальную систему.

Консенсус может быть реализован разными способами, например следующими:

- с помощью алгоритмов на основе лотереи, включая методы на основе доказательства потраченного времени (Proof of Elapsed Time, PoET) и доказательства работы (PoW);
- с помощью методов на основе голосования, включая задачу на основе византийских генералов с избыточностью (Redundant Byzantine Fault Tolerance, RBFT) и Paxos (позволяет достичь консенсуса в распределенной системе с отказами узлов); каждый из этих подходов нацелен на различные сетевые требования и модели отказоустойчивости.

Алгоритмы на основе лотереи выгодны тем, что они могут масштабироваться до большого числа узлов, поскольку победитель лотереи предлагает блок и передает

его остальной части сети для проверки. С другой стороны, эти алгоритмы могут привести к разветвлению, когда два «победителя» предлагают свои блоки. Каждое разветвление должно быть разрешено, что увеличивает время до завершения.

Алгоритмы на основе голосования выгодны тем, что обеспечивают завершение с малой задержкой.

Когда большинство узлов проверяют транзакцию или блок, существует консенсус и наступает завершение. Поскольку алгоритмы на основе голосования обычно требуют, чтобы узлы передавали сообщения каждому из других узлов в сети, чем больше узлов существует в сети, тем больше времени требуется для достижения консенсуса. Это приводит к компромиссу между масштабируемостью и скоростью.

Разработчики Hyperledger исходят из того, что сети бизнес-блокчейнов будут работать в среде частичного доверия. Учитывая это, в системе не используются стандартные консенсусные подходы PoW с анонимными майнерами. По оценке разработчиков, эти подходы требуют слишком больших затрат с точки зрения ресурсов и времени, чтобы быть оптимальными для бизнес-сетей блокчейнов [148].

4.3.2 Проекты экосистемы Hyperledger

Hyperledger Foundation поддерживает ряд проектов программного обеспечения блокчейн-систем корпоративного уровня. Проекты задуманы и созданы сообществом разработчиков для поставщиков услуг, организаций конечных пользователей, стартапов, ученых и других заинтересованных лиц и организаций, чтобы использовать их для создания и развертывания сетей цепочек блоков или коммерческих решений.

Команда Hyperledger Foundation является частью более крупной команды Linux Foundation, имеющей многолетний опыт в предоставлении услуг по управлению программами для проектов с открытым исходным кодом.

Все проекты экосистемы Hyperledger следуют единой философии, имеющей следующие отличительные черты:

- модульность;
- высокая безопасность;
- совместимость;
- независимость от криптовалюты;
- наличие API.

Экосистема Hyperledger включает в себя множество фреймворков и библиотек; рассмотрим основные из них далее [149].

Hyperledger Burrow – это полный блокчейн-дистрибутив, ориентированный на простоту, скорость и эргономичность разработчика.

Он поддерживает смарт-контракты на основе EVM и WASM (WebAssembly – технология выполнения в веб-приложениях байт-кода, написанного на различных языках программирования) и использует консенсус BFT через алгоритм Tendermint (см. [54]). Burrow имеет сложную систему событий и может поддерживать отображение данных в цепочке в реляционной базе данных.

Hyperledger Fabric предназначена в качестве основы для разработки приложений или решений с модульной архитектурой с использованием технологии распределенного реестра. Hyperledger Fabric позволяет использовать такие компоненты, как консенсус и услуги членства, по принципу «подключи и ра-

ботай». Ее модульная и универсальная конструкция удовлетворяет широкому спектру сценариев использования.

Hyperledger Fabric поддерживает разработку смарт-контрактов на различных языках программирования, включая golang, Java, JavaScript, TypeScript.

Hyperledger Grid – это платформа для построения решений цепочки поставок, включающих компоненты распределенного реестра.

Hyperledger Indy предоставляет инструменты, библиотеки и компоненты многократного использования для предоставления цифровых удостоверений, основанных на блокчейн-цепочках или других распределенных реестрах, чтобы они могли взаимодействовать между административными доменами, приложениями и любым другим изолированным хранилищем. Indy совместим с другими блокчейн-системами или может использоваться автономно.

Hyperledger Iroha спроектирован таким образом, чтобы его можно было просто и легко включить в инфраструктурные проекты или проекты интернета вещей, требующие наличия распределенного реестра. Hyperledger Iroha отличается простой конструкцией, модульной архитектурой, построенной с использованием языка C++ на основе предметной области, акцентом на разработку клиентских приложений и новым отказоустойчивым консенсусным алгоритмом, который называется YAC (Yet Another Consensus – «Еще один консенсус»).

Hyperledger Sawtooth предлагает гибкую и модульную архитектуру, отделяющую базовую систему от домена приложения, поэтому смарт-контракты могут определять бизнес-правила для приложений без необходимости знать дизайн базовой системы. Hyperledger Sawtooth поддерживает множество консенсусных алгоритмов, включая практическую византийскую отказоустойчивость (PBFT) и доказательство потраченного времени (PoET).

Hyperledger Caliper – фреймворк для разработки блокчейн-сети, позволяет мониторить ресурсы сети.

Hyperledger Cello представляет собой сервис для деплоя смарт-контракта в реестр.

Hyperledger Composer – обозреватель различных данных в блокчейн-системе. Позволяет удобно работать с транзакциями. На данный момент устарел и не поддерживается.

Hyperledger Explorer выполняет, по сути, те же функции, что и Composer. На данный момент поддерживается и активно развивается.

Hyperledger Quilt – это Java-реализация Interledger – набора открытых протоколов и стандартов, который обеспечивает совместимость платежей в любой валюте – фиатной или криптовалютной. На данный момент не поддерживается.

Hyperledger Ursula – криптографическая библиотека, помогает работать с различными криптографическими алгоритмами.

4.3.3 Архитектура Hyperledger Fabric

Опишем ключевые понятия архитектуры Hyperledger Fabric, которые позволяют платформе быть готовым, но настраиваемым промышленным блокчейн-решением:

- активы. Возможность определения своих активов позволяет обмениваться практически всем, что имеет денежную стоимость в сети: от продуктов питания и антикварных автомобилей до валютных фьючерсов;

- чейнкоды. Чейнкод выполняется отдельно от транзакций, что понижает требуемый уровень доверия и проверки между разными типами узлов, а также оптимизирует производительность и масштабируемость;
- реестр. Неизменяемый распределенный реестр сохраняет всю историю транзакций каждого канала и включает в себя возможность совершения SQL-подобных запросов, что обеспечивает простое и эффективное проведение аудитов, а также однозначное разрешение споров;
- конфиденциальность. Каналы (представляющие собой выделенные виртуальные сети для взаимодействия между определенным подмножеством пользователей) и коллекции конфиденциальных данных позволяют осуществлять конфиденциальные и многосторонние сделки, что обычно требуется для сети, в которую входят конкурирующие между собой предприятия, или для сети определенной регулируемой отрасли;
- консенсус. Уникальный подход к консенсусу обеспечивает необходимые для участников системы гибкость и масштабируемость.

Активы могут варьироваться от материального (недвижимость и оборудование) до нематериального имущества (контракты и интеллектуальная собственность). Hyperledger Fabric позволяет манипулировать активами с помощью чейнкод-транзакций.

Активы реализованы в Hyperledger Fabric как коллекция пар «ключ–значение», с изменениями состояния, записанными в качестве транзакций в реестре канала. Активы могут быть представлены в двоичном и/или JSON-формате.

Чейнкод в данном случае представляет собой программу, определяющую активы и инструкции по изменению активов; можно сказать, чейнкод определяет бизнес-логику работы с активами. Чейнкод обеспечивает соблюдение правил чтения и изменения пар «ключ–значение» или другой информации из базы данных.

Чейнкод исполняется на основе текущего состояния базы данных реестра и инициализируется предложением о проведении транзакции (*transaction proposal*). Результатом работы чейнкода является набор записей (*write set*) пар «ключ–значение», которые могут быть занесены в реестр всех узлов и представлены в сеть.

Реестр – это последовательная, защищенная от подделки запись всех переходов состояния Fabric. Переходы состояния являются результатом вызова чейнкода (транзакций) участвующими сторонами. Результатом каждой транзакции является набор привязанных к реестру пар «ключ–значение», которые удаляются, создаются или обновляются.

Реестр состоит из блокчейн-цепочки для хранения неизменяемой последовательности блоков, а также из базы данных состояний, поддерживающей текущие состояния Fabric. Для каждого канала существует свой реестр и своя база данных состояний. Каждый узел поддерживает копию реестра каждого канала, в котором он состоит.

4.3.4 Пример смарт-контракта для Hyperledger

Приведем пример последовательности действий, включающих в себя подготовку операционной системы, настройку локального окружения и разработку простого смарт-контракта для системы Hyperledger.

Подготовка и настройка

Опишем порядок установки требуемых компонентов в операционной системе Windows.

На первом шаге необходимо убедиться, что используемая версия Windows поддерживает установку подсистемы Windows для Linux (WSL).

Для этого следует проверить версию Windows и номер сборки. Сделать это можно путем одновременного нажатия клавиш «Windows» и «R» и последующего запуска программы winver.

Для использования WSL необходимо наличие Windows 10 сборки не ниже 19041 или Windows 11 [180].

На втором шаге включим подсистему Windows для поддержки ОС Linux, что необходимо для последующей установки ОС Linux в Windows.

Сделать это можно одним из следующих способов:

- запустить PowerShell с правами администратора и выполнить следующую команду:

```
dism.exe/online/enable-feature/featurename:Microsoft-Windows-Subsystem-Linux/all/norestart
```

- зайти в раздел «Программы и компоненты» панели управления, выбрать пункт «Включение или отключение компонентов Windows» и включить компонент «Подсистема Windows для Linux».

На третьем шаге следует включить компонент поддержки виртуальных машин путем запуска PowerShell с правами администратора и выполнения следующей команды:

```
dism.exe/online/enable-feature/featurename:VirtualMachinePlatform/all/norestart
```

После этого необходимо перезапустить компьютер для завершения установки и обновления WSL до WSL 2.

На четвертом шаге выберем используемую по умолчанию версию WSL с помощью следующей команды:

```
wsl -set default-version 2
```

На следующем шаге установим дистрибутив Ubuntu Linux, который можно скачать в Microsoft Store.

Затем необходимо установить Docker [141] – программное обеспечение, позволяющее запускать приложения в изолированных контейнерах. Оно полезно, когда нет необходимости разворачивать виртуальную машину ради некоторого приложения. В Docker приложение будет работать в основной системе Windows и использовать компоненты, предназначенные для Linux.

При первом запуске Docker нужно зарегистрироваться, зайти в настройки ПО, раздел «Sources», далее выбрать «WSL integration». В приведенном списке появится Ubuntu, которую следует включить и сохранить изменения. Это необходимо для того, чтобы подсистема Ubuntu увидела Docker.

Для завершения настройки выполним следующие команды в Linux:

```
sudo apt-get update
sudo apt-get install curl
sudo chmod 666 /var/run/docker.sock
```

Кроме того, необходимо установить Node.js и менеджер пакетов NPM (Node Package Manager):

```
sudo apt install nodejs  
sudo apt install npm
```

Развертывание тестовой сети

Скачаем примеры проектов и тестовую сеть, для этого выполним команду в терминале Linux, после чего загрузится папка в текущую директорию и произойдет скачивание конфигурационной блокчейн-сети для развертывания тестовой сети:

```
curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Загруженная система Hyperledger содержит следующие компоненты:

- Fabric-tools – инструменты разработки;
- Fabric-peers – узлы.

То есть в результате загружаются и примеры смарт-контрактов, и тестовая сеть. Если мы хотим заниматься только разработкой смарт-контракта, то мы можем развернуть тестовую сеть буквально парой команд. Именно поэтому используется Docker, он поднимает несколько узлов сети (несколько рабочих станций).

Когда мы скачали Hyperledger, то в консоли увидим уже скачанные образы. Далее переходим в папку test-network, из нее в дальнейшем будем поднимать тестовую сеть.

Запуск тестовой сети выполняется следующей командой:

```
./network.sh up
```

Затем для функционирования контракта необходимо создать канал:

```
./network.sh createChannel
```

По умолчанию канал создается с именем mychannel. Если необходимо указать другое имя канала, то выполняем эту же команду с флагом -с и именем канала.

После окончания работы с сетью для ее удаления необходимо выполнить команду:

```
./network.sh down
```

Чтобы задеплоить смарт-контракт в сеть, используется следующая команда:

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-javascript -ccl javascript
```

Опишем основные параметры данной команды:

- -ccn указывает на то, что код необходимо упаковать в архив с именем basic;
- -ccp указывает на то, в какой папке лежит исходный код смарт-контракта;
- -ccl указывает, на каком языке написан смарт-контракт.

В папке test-network репозитория Hyperledger находится файл network.sh, в котором расписаны все функции для взаимодействия с сетью: запуск сети, добавление узла, удаление узла, отключение сети, генерация сертификата, деплой смарт-контракта и т. д.

При рассмотрении этого сложного файла возникает вопрос: неужели все это нужно прописывать снова при запуске, например, какого-то нового коммерческого проекта? На самом деле нет: если мы используем какое-то облачное решение, допустим IBM, то компания – разработчик решения предоставляет готовый API – Blockchain SDK, с помощью которого можно намного проще и быстрее построить сеть через графическую оболочку.

Пример смарт-контракта

В папке asset-transfer-basic (которая находится в папке fabric-samples) можно увидеть пример смарт-контракта на передачу какого-либо имущества от одного собственника к другому. Рассмотрим далее реализацию с использованием языка программирования JavaScript в папке chaincode-javascript.

В файле index.js инициализируется и экспортится контракт.

Сама логика смарт-контракта находится в папке lib в файле assetTransfer.js.

Рассмотрим основные функции данного контракта.

Функция создания транзакции на вход принимает параметры транзакции, затем формируется ассоциативный массив и с помощью метода putState заносится в блокчейн (рис. 4.77).

```

72     // CreateAsset issues a new asset to the world state with given details.
73     async CreateAsset(ctx, id, color, size, owner, appraisedValue) {
74         const exists = await this.AssetExists(ctx, id);
75         if (exists) {
76             throw new Error(`The asset ${id} already exists`);
77         }
78
79         const asset = {
80             ID: id,
81             Color: color,
82             Size: size,
83             Owner: owner,
84             AppraisedValue: appraisedValue,
85         };
86         //we insert data in alphabetic order using 'json-stringify-deterministic' and 'sort-keys-recursive'
87         await ctx.stub.putState(id, Buffer.from(stringify(sortKeysRecursive(asset))));
88         return JSON.stringify(asset);
89     }

```

Рисунок 4.77. Функция создания транзакции

Функция получения транзакции по идентификатору получает на вход в качестве параметра идентификатор и, если транзакция существует, возвращает содержимое этой транзакции (рис. 4.78).

```

92     async ReadAsset(ctx, id) {
93         const assetJSON = await ctx.stub.getState(id); // get the asset from chaincode state
94         if (!assetJSON || assetJSON.length === 0) {
95             throw new Error(`The asset ${id} does not exist`);
96         }
97         return assetJSON.toString();
98     }

```

Рисунок 4.78. Функция получения транзакции по идентификатору

Функция обновления транзакции получает на вход идентификатор существующей транзакции, а также поля, которые необходимо изменить, далее на основе этих полей формируется массив и заносится в блокчейн (рис. 4.79). Если необходимо изменить только один параметр транзакции, то мы передаем лишь его.

```

101    async UpdateAsset(ctx, id, color, size, owner, appraisedValue) {
102        const exists = await this.AssetExists(ctx, id);
103        if (!exists) {
104            throw new Error(`The asset ${id} does not exist`);
105        }
106
107        // overwriting original asset with new asset
108        const updatedAsset = {
109            ID: id,
110            Color: color,
111            Size: size,
112            Owner: owner,
113            AppraisedValue: appraisedValue,
114        };
115        // we insert data in alphabetic order using 'json-stringify-deterministic' and 'sort-keys-recursive'
116        return ctx.stub.putState(id, Buffer.from(stringify(sortKeysRecursive(updatedAsset))));
117    }

```

Рисунок 4.79. Функция обновления транзакции

Функция передачи прав другому владельцу реализовывается так же, как и функция обновления, только мы меняем всего лишь одно поле Owner.

В папке asset-transfer-basic, помимо смарт-контракта, есть еще и пример простой консольной реализации клиентского приложения, он находится в папке application-javascript.

Чтобы подключиться к блокчейн-сети, необходимо в приложение передать следующую информацию: название канала, название проекта (его указывали при деплее контракта в сеть), название организации и имя пользователя, а также его сертификат (рис. 4.80).

```

15  const channelName = 'mychannel';
16  const chaincodeName = 'basic';
17  const mspOrg1 = 'Org1MSP';
18  const walletPath = path.join(__dirname, 'wallet');
19  const org1UserId = 'appUser';

```

Рисунок 4.80. Параметры для подключения к блокчейн-сети

Если мы вызываем функцию контракта, меняющую данные блокчейна, то используем метод `submitTransaction`, который в качестве первого параметра принимает название функции смарт-контракта, а затем параметры этой функции (рис. 4.81).

```
console.log(`\n--> Submit Transaction: CreateAsset, creates new asset with ID, color, owner, size, and appraisedValue arguments`);
result = await contract.submitTransaction('CreateAsset', 'asset13', 'yellow', '5', 'Tom', '1300');
console.log(`*** Result: committed`);
if (`${result}` !== '') {
  console.log(`*** Result: ${prettyJSONString(result.toString())}`);
}
```

Рисунок 4.81. Вызов функции создания транзакции

Для вызова функции смарт-контракта, которая не меняет данные в цепочке (например, просмотр), используется метод `evaluateTransaction`, передача параметров в который аналогична методу `submitTransaction` (рис. 4.82).

```
console.log(`\n--> Evaluate Transaction: ReadAsset, function returns an asset with a given assetID`);
result = await contract.evaluateTransaction('ReadAsset', 'asset13');
console.log(`*** Result: ${prettyJSONString(result.toString())}`);
```

Рисунок 4.82. Вызов функции получения транзакции по идентификатору

На примере тестового смарт-контракта мы рассмотрели основные методы смарт-контрактов и способы их использования в простом клиентском приложении в системе Hyperledger.

Тестовые смарт-контракты можно использовать для разработки собственных смарт-контрактов.

4.4 Обзор других платформ

4.4.1 EOSIO

EOSIO – это блокчейн-платформа, разработанная как для общедоступных, так и для частных случаев использования. Она настраивается в соответствии с широким спектром бизнес-потребностей в различных отраслях с широкими полномочиями безопасности на основе ролей, лучшей в отрасли скоростью и безопасной обработкой приложений.

При создании EOSIO применяются знакомые шаблоны разработки и языки программирования, используемые существующими приложениями, не связанными с блокчейном, поэтому разработчики могут создавать без проблем пользовательский интерфейс с помощью инструментов разработки, которые они уже знают и любят.

С помощью EOSIO компании из множества секторов и отраслей быстро движутся к созданию децентрализованной площадки для торговли и коммуникаций.

На блокчейн-платформах, использующих механизм консенсуса PoW, взимается комиссия за каждую обработанную транзакцию. EOSIO устраняет комиссии, возлагая на сеть бремя этих затрат за счет инфляции, вместо того чтобы заставлять отдельных пользователей сети платить за доступ к ее объединен-

ным ресурсам. Другими словами, владение токенами покрывает любые сопутствующие расходы.

В EOSIO держатели токенов могут выбрать 21 производителя блоков для обработки транзакций, что распределяет власть между держателями токенов и согласовывает интересы всех вовлеченных сторон. Самая большая разница между EOSIO и другими блокчейн-протоколами заключается в степени децентрализации.

Платформа EOSIO предоставляет такие функции, как учетные записи, аутентификация, базы данных, асинхронный обмен данными и выполнение приложений на нескольких вычислительных ядрах и кластерах. Эти функции также распространены в средах разработки программного обеспечения, не основанных на блокчейн-технологиях.

В экосистеме EOSIO создание блоков и проверка блоков выполняются специальными узлами, называемыми «производителями блоков». Производители избираются заинтересованными сторонами EOSIO.

Каждый производитель запускает экземпляр узла EOSIO через службу nodeos. По этой причине производители, которые находятся в активном расписании (см. далее) для производства блоков, также называются «активными», или «производящими», узлами.

Блокчейны на основе EOSIO используют делегированное подтверждение доли (DroS) для выбора активных производителей, которым будет разрешено подписывать действительные блоки в сети. Однако это только половина процесса консенсуса EOSIO.

Другая половина участвует в фактическом процессе подтверждения каждого блока до тех пор, пока он не станет окончательным (необратимым), который выполняется асинхронным отказоустойчивым способом на основе задачи византийских генералов (Asynchronous Byzantine Fault Tolerance, ABFT). Следовательно, в модели консенсуса EOSIO задействованы два уровня:

- уровень 1 – модель собственного согласия (ABFT);
- уровень 2 – делегированное подтверждение ставки (DroS).

Фактическая собственная модель консенсуса, используемая в EOSIO, не имеет концепции делегирования/голосования, доли или даже токенов. Они используются уровнем DroS для генерации первого расписания производителей блоков и, если применимо, обновления набора не более чем на каждом цикле расписания, после того как каждый производитель прошел цикл. Эти два уровня функционально разделены в программном обеспечении EOSIO.

Первый уровень решает, какие блоки, полученные и синхронизированные между выбранными производителями, в конечном итоге станут окончательными и, следовательно, навсегда сохранятся в цепочке блоков. Он получает расписание производителей, предложенное вторым уровнем, и использует это расписание, чтобы определить, какие блоки правильно подписаны соответствующим производителем.

Для византийской отказоустойчивости уровень использует двухэтапный процесс подтверждения блока, при котором подавляющее большинство в две трети производителей из текущего запланированного набора подтверждает каждый блок дважды.

На первом этапе подтверждения предлагается последний необратимый блок (Last Irreversible Block, LIB). Второй этап подтверждает предлагаемый LIB как окончательный. Этот уровень также используется для сигнализации об изменениях расписания производителя, если таковые имеются, в начале каждого цикла расписания.

Слой DpoS (уровень 2) управляет вопросами токенов, ставок, голосования и делегирования, подсчета голосов, ранжирования производителей и инфляционного вознаграждения. Этот уровень также отвечает за генерацию новых расписаний производителей на основе рейтингов, созданных в результате голосования производителей.

Это происходит в циклах расписания продолжительностью примерно две минуты (126 секунд), что является периодом, который требуется производителю блоков, чтобы назначить временной интервал для создания и подписи блоков. Временной интервал длится в общей сложности 6 секунд на производителя, что является раундом производителя, в течение которого может быть создано и подписано максимум 12 блоков [115].

4.4.2 Краткий обзор прочих блокчейн-платформ

В данном разделе рассмотрим существующие платформы (помимо описанных ранее Эфириум и Hyperledger), которые позволяют разрабатывать приложения на базе блокчейн-технологий.

MultiChain

Платформа с открытым исходным кодом MultiChain позволяет пользователям развертывать частную блокчейн-сеть для проектов, в которых осуществляются финансовые операции. MultiChain дает возможность строить сеть как внутри какой-либо организации, так и между несколькими организациями.

MultiChain поддерживает выполнение программного обеспечения системы на серверах под управлением Windows, Linux и Mac OS и предоставляет простой интерфейс разработки приложений (API) и интерфейс командной строки.

Основные задачи, которые решает платформа MultiChain:

- обеспечение видимости работы только участникам приватной сети;
- контроль за разрешенными транзакциями;
- обеспечение майнинга без консенсуса PoW.

Проблемы, связанные с масштабированием сети, легко решить за счет того, что сеть является приватной и участники могут регулировать размер блока.

В MultiChain все привилегии предоставляются и отменяются с помощью сетевых транзакций, содержащих специальные метаданные. Майнер первого блока («генезис-блока») автоматически получает все привилегии, в том числе права администратора для управления привилегиями других пользователей [185].

В MultiChain плата за транзакцию и поощрение за генерацию блока по умолчанию равны нулю. Однако можно указать какие-либо значения данных параметров в файле params.dat. Этот файл содержит всю конфигурацию системы, включая следующие данные:

- протокол формирования цепочки;
- целевое время для генерации блока;

- тип активного уровня доступа;
- тип майнинга (с доказательством работы или без него);
- стимул майнинга;
- вид сделки;
- максимальный размер блока;
- максимальное количество метаданных за транзакцию.

Ripple

В отличие от рассмотренной в предыдущем разделе платформы MultiChain, платформа Ripple позволяет строить приложения с использованием блокчейн-технологий на базе криптовалюты Ripple – XRP. Ripple представляет собой децентрализованную блокчейн-платформу с открытым исходным кодом, которая может выполнять транзакции за 3–5 секунд.

Ripple представляет собой открытую платформу: разработчики могут использовать XRP и лежащие в его основе технологии в различных проектах от микроплатежей до электронной коммерции, бирж и одноранговых сервисов [209].

Stellar

Stellar представляет собой сеть с открытым исходным кодом для валют и платежей. Stellar позволяет создавать, отправлять и торговать различными цифровыми активами, включая валюты: доллары, песо, биткойны и др.

Данная платформа разработана таким образом, чтобы все финансовые системы мира (как классические, так и криптовалютные) могли работать вместе в одной сети [229].

Изначально платформа Stellar была ответвлением платформы Ripple, но в дальнейшем был разработан собственный протокол с открытым исходным кодом.

Corda

Corda является блокчейн-платформой на основе программного обеспечения с открытым исходным кодом с 2016 года.

Разработка Corda была основана на сотрудничестве глобального консорциума организаций, представляющих многие отрасли, и участие регулирующих органов было ключевым элементом этого процесса разработки. Требования финансовой индустрии легли в основу архитектуры Corda, но практический опыт показал, что Corda имеет более широкое применение, выходящее далеко за рамки банковского дела.

Основная проблема, которую Corda стремится решить, – это проблема управления контрактами и другими соглашениями между любым сочетанием фирм и частных лиц, особенно когда эти стороны доверяют друг другу достаточно, чтобы торговаться, но недостаточно, чтобы их контрагент вел все записи [234].

В Corda два основных аспекта консенсуса:

- 1) действительность транзакции: стороны могут достичь уверенности в том, что предложенная транзакция действительна, путем проверки того, что связанный код контракта, который должен быть детерминированным, успешно проверяется и имеет все необходимые подписи и что любые транзакции, к которым относится эта транзакция, также действительны;

- 2) уникальность транзакции: стороны могут быть уверены в том, что рассматриваемая транзакция является уникальным потребителем всех своих входных состояний. То есть не существует другой транзакции, по которой ранее был достигнут консенсус, которая потребляет любое из тех же состояний.

Neo

Платформа Neo, как и подробно рассмотренная ранее платформа Эфириум, поддерживает смарт-контракты.

Neo имеет две формы цифровых активов: глобальные активы и контрактные активы.

- Глобальные активы могут быть записаны в системном пространстве и могут быть идентифицированы всеми смарт-контрактами и клиентами.
- Контрактные активы записываются в частном хранилище смарт-контракта и требуют, чтобы совместимый клиент распознал их; контрактные активы могут соответствовать определенным стандартам, чтобы обеспечить совместимость с большинством клиентов.

В Neo реализован набор стандартов цифровой идентификации, совместимых с X.509 и соответствующих концепции инфраструктуры открытых ключей PKI (Public Key Infrastructure). При этом вместо применяемых в PKI списков отозванных сертификатов (Certificate Revocation List, CRL) или протокола определения состояния сертификатов OCSP (Online Certificate Status Protocol) используются возможности блокчейн-сети.

Система смарт-контрактов NeoContract – важнейшая особенность платформы Neo, обеспечивающая ее бесшовную интеграцию в существующие экосистемы разработчиков. Разработчикам не нужно изучать новый язык программирования, они используют C#, Java и другие распространенные языки программирования в своих знакомых средах IDE (Visual Studio, Eclipse и т. д.) для разработки, отладки и компиляции смарт-контрактов. NeoContract позволяет разработчикам быстро осуществлять разработку смарт-контрактов.

Еще один компонент платформы – универсальная легкая виртуальная машина Neo, NeoVM, которая обладает такими преимуществами, как высокая надежность, высокий уровень параллелизма и высокая масштабируемость.

В качестве метода обеспечения консенсуса платформа Neo использует рассмотренный ранее механизм на основе делегированной задачи византийских генералов DBFT. В марте 2019 года была выпущена обновленная версия DBFT 2.0, которая повышает надежность и безопасность за счет введения трехэтапного консенсуса, а также механизма восстановления [189].

4.4.3 Обзор отечественных решений

Кратко опишем в качестве примера некоторые из существующих российских решений на базе распределенных реестров.

«Мастерчейн» ассоциации «ФинТех»

В 2016 году Банк России анонсировал запуск разработки отечественного решения на базе технологий распределенного реестра. Основные партнеры проекта – крупнейшие финансовые организации России, входящие в ассоциацию

«ФинТех», среди них крупнейшие банки страны, платежные системы, а также провайдеры связи и мобильные операторы [36, 41].

«Мастерчайн» разрабатывался как крупнейшая распределенная система для использования всеми финансовыми организациями страны с целью формирования прозрачной отчетности, упрощения существующих сервисов и формирования новых.

Система является модернизированным аналогом платформы Эфириум и поддерживает весь необходимый функционал – идентификацию пользователей по открытому ключу, использование ЭП и, конечно, выполнение смарт-контрактов. Все операции с данными жестко сконфигурированы в соответствии с разработанной ролевой системой доступа к информации.

Оператором платформы «Мастерчайн» является компания «Цифровые распределенные реестры».

В 2019 году платформа успешно прошла экспертизу ФСБ на используемые средства криптографической защиты информации (СКЗИ). Платформа соответствует всем требованиям отечественных регуляторов в области защиты информации, в том числе в части использования стандартов шифрования и ЭП. А отсутствие в информационной системе персональных данных и сведений, составляющих коммерческую тайну, облегчило эту задачу.

Платформа включает в себя шесть основных сервисов, которые обеспечивают функционал по работе с финансовыми цифровыми активами, в том числе сервисы банковских гарантий и поддержки цифровых кредитивов.

Несмотря на все преимущества платформы и ее успешное развитие по настоящее время, есть ряд сложностей. В частности, в 2019 году один из активных участников платформы – Сбербанк – заявил о выходе из проекта в связи с превышением централизации, сославшись на то, что прозрачность системы реализована только для руководителей и операторов системы, что нарушает основные принципы блокчейн-технологий. Кроме того, была обозначена проблема быстродействия системы – система проводила транзакции в течение 30 секунд, что не соответствует запросам организации к платформе. После чего Сбербанк перешел к использованию платформ на базе описанной ранее Hyperledger Fabric [246].

Waves Enterprise

В 2016 г. была также основана компания Waves, которая на сегодняшний день создала большое количество блокчейн-решений не только для российского, но и для международного рынка. Успешный старт с помощью краудфандинга позволил компании развиваться высокими темпами.

В 2017 году компания реализует интеграцию в большое количество цифровых сервисов и занимает значимое место на рынке. В основе платформы лежит консенсус Leased Proof of Stake (LPoS – арендованное доказательство доли владения), что являлось новинкой того времени и до сих пор привлекает большое количество пользователей.

На сегодняшний день в портфолио Waves присутствуют сервисы для организации голосований, системы защиты авторского права, медицинские карты пациентов на базе распределенных реестров, интеграции в системы интернета вещей, система документооборота, сервисы контроля цепочек поставок и, конечно, системы финансового обращения [10].

Российские системы электронного голосования

В конце 2020 года Лаборатория Касперского выпустила систему электронного голосования, которая получила название Polys. Участник голосования подтверждал свою личность и получал доступ к голосованию через анонимный QR-код. Голосование доступно в специальных терминалах, а также с помощью мобильных устройств [199].

На базе Polys Департамент информационных технологий Москвы разработал первую систему электронного голосования, которая применялась во время официального голосования – по поправкам в Конституцию РФ в 2020 году. Проект был пилотно запущен в Московской и Нижегородской областях вместе с возможностью участия в традиционном голосовании с бумажными бюллетенями.

В 2020 году Центр технологий распределенных реестров Санкт-Петербургского государственного университета выпускает в свет платформу «Криптовече» – корпоративный сервис для проведения различного рода голосований – на заседаниях и собраниях, выборах на различные должности в организациях. Платформа основана на базе Hyperledger Fabric [23].

Ранее мы уже упомянули компанию Waves как разработчика системы голосования. Так же, как и «Криптовече», Waves предоставляет удобный сервис We.Vote, создавая конкуренцию на рынке. Платформа подходит для тех же целей – корпоративных голосований и выборов [18].

Однако этим опыт компании не ограничивается, в 2020 году она совместно с компанией Ростелеком по заказу Центральной избирательной комиссии готовила систему дистанционного электронного голосования, которая использовалась на дополнительных выборах депутатов Государственной думы седьмого созыва в Курской и Ярославской областях, а также при электронном голосовании на выборах в Государственную Думу восьмого созыва в 2021 г.

Все упомянутые выше системы электронного голосования соответствуют всем основным требованиям – обеспечить тайну голосования каждого пользователя, гарантировать однократное участие избирателя в голосовании, а также обеспечить надежный подсчет голосов для обеспечения легитимности избирательного процесса и его результатов.

Промышленные криптосистемы

В 2017 году свою работу над смарт-контрактами и блокчейн-системами начало Конструкторское бюро «Контракт», позже присоединившееся к группе компаний «ИнфоТеКС» под именем «ПроКСи» («Промышленные криптосистемы»).

За 5 лет работы компания создала несколько успешных проектов и внедрила их в промышленном комплексе нашей страны. Один из наиболее популярных проектов – система Smart Fuel – предоставляет возможность мгновенного безналичного расчета за топливо; теперь транзакции проходят за 15 секунд, что на порядки быстрее, чем это было ранее. Платформа была разработана по заказу Газпромнефти, и на текущий момент все больше аэропортов и топливо-заправочных станций подключаются к системе.

Другое успешное решение – ФГИС «Семеноводство» – обеспечивает контроль за посевным материалом и налаживает отношения между заказчиком и поставщиком партии, контролируя процесс поставки.

Кроме того, по заказу РЖД компания разработала систему смарт-контрактов сервисного обслуживания (СКСО), контролирующую процесс технического обслуживания локомотивов, их эксплуатационные характеристики и финансовые операции с сервисными центрами.

Все сервисы компании «ПроКСи» базируются на платформе блокчейна Hyperledger Fabric [40].

Приложение 1

Таблицы констант алгоритмов хеширования

Алгоритмы хеширования, описание которых было приведено в главе 1 книги, используют в процессе преобразований различные константы. Значения таких констант приведены в настоящем приложении.

П1.1 Таблица замен алгоритма MD2

Таблица П1.1. Таблица замен алгоритма MD2

41	46	67	201	162	216	124	1
61	54	84	161	236	240	6	19
98	167	5	243	192	199	115	140
152	147	43	217	188	76	130	202
30	155	87	60	253	212	224	22
103	66	111	24	138	23	229	18
190	78	196	214	218	158	222	73
160	251	245	142	187	47	238	122
169	104	121	145	21	178	7	63
148	194	16	137	11	34	95	33
128	127	93	154	90	144	50	39
53	62	204	231	191	247	151	3
255	25	48	179	72	165	181	209
215	94	146	42	172	86	170	198
79	184	56	210	150	164	125	182
118	252	107	226	156	116	4	241
69	157	112	89	100	113	135	32

Окончание табл. П1.1

134	91	207	101	230	45	168	2
27	96	37	173	174	176	185	246
28	70	97	105	52	64	126	15
85	71	163	35	221	81	175	58
195	92	249	206	186	197	234	38
44	83	13	110	133	40	132	9
211	223	205	244	65	129	77	82
106	220	55	200	108	193	171	250
36	225	123	8	12	189	177	74
120	136	149	139	227	99	232	109
233	203	213	254	59	0	29	57
242	239	183	14	102	88	208	228
166	119	114	248	235	117	75	10
49	68	80	180	143	237	31	26
219	153	141	51	159	17	131	20

Каждая n -я ячейка приведенной выше таблицы содержит значение $S[n]$, $0 \leq n \leq 255$.

П1.2 Индексы используемых в итерациях слов блока сообщения алгоритма MD4

Таблица П1.2. Индексы используемых слов блока сообщения в зависимости от номера итерации алгоритма MD4

Номер итерации j	x_j
0, 16, 32	0
1, 20, 40	1
2, 24, 36	2
3, 28, 44	3
4, 17, 34	4
5, 21, 42	5
6, 25, 38	6

Окончание табл. П1.2

Номер итерации j	x_j
7, 29, 46	7
8, 18, 33	8
9, 22, 41	9
10, 26, 37	10
11, 30, 45	11
12, 19, 35	12
13, 23, 43	13
14, 27, 39	14
15, 31, 47	15

П1.3 Константы алгоритма MD5

Константы, используемые алгоритмом MD5, приведены в табл. П1.3–П1.5.

Таблица П1.3. Индексы используемых слов блока сообщения в зависимости от номера итерации алгоритма MD5

Номер итерации j	x_j
1, 20, 42, 49	0
2, 17, 37, 56	1
3, 30, 48, 63	2
4, 27, 43, 54	3
5, 24, 38, 61	4
6, 21, 33, 52	5
7, 18, 44, 59	6
8, 31, 39, 50	7
9, 28, 34, 57	8
10, 25, 45, 64	9
11, 22, 40, 55	10
12, 19, 35, 62	11
13, 32, 46, 53	12
14, 29, 41, 60	13
15, 26, 36, 51	14
16, 23, 47, 58	15

Таблица П1.4. Модифицирующие константы в зависимости от номера итерации алгоритма MD5

Номер итерации j	T_j
1	D76AA478
2	E8C7B756
3	242070DB
4	C1BDCEEE
5	F57C0FAF
6	4787C62A
7	A8304613
8	FD469501
9	698098D8
10	8B44F7AF
11	FFFF5BB1
12	895CD7BE
13	6B901122
14	FD987193
15	A679438E
16	49B40821
17	F61E2562
18	C040B340
19	265E5A51
20	E9B6C7AA
21	D62F105D
22	02441453
23	D8A1E681
24	E7D3FBC8
25	21E1CDE6
26	C33707D6
27	F4D50D87
28	455A14ED

Продолжение табл. П1.4

Номер итерации j	T_j
29	A9E3E905
30	FCEFA3F8
31	676F02D9
32	8D2A4C8A
33	FFFFA3942
34	8771F681
35	6D9D6122
36	FDE5380C
37	A4BEEA44
38	4BDECFA9
39	F6BB4B60
40	BEBFBC70
41	289B7EC6
42	EAA127FA
43	D4EF3085
44	04881D05
45	D9D4D039
46	E6DB99E5
47	1FA27CF8
48	C4AC5665
49	F4292244
50	432AFF97
51	AB9423A7
52	FC93A039
53	655B59C3
54	8F0CCC92
55	FFEFFF47D
56	85845DD1
57	6FA87E4F

Окончание табл. П1.4

Номер итерации j	T_j
58	FE2CE6E0
59	A3014314
60	4E0811A1
61	F7537E82
62	BD3AF235
63	2AD7D2BB
64	EB86D391

Модифицирующие константы в табл. П1.4 приведены в шестнадцатеричном виде.

Таблица П1.5. Количество битов циклического сдвига влево в зависимости от номера итерации алгоритма MD5

Номер итерации j	s_j
33, 37, 41, 45	4
17, 21, 25, 29	5
49, 53, 57, 61	6
1, 5, 9, 13	7
18, 22, 26, 30	9
50, 54, 58, 62	10
34, 38, 42, 46	11
2, 6, 10, 14	12
19, 23, 27, 31	14
51, 55, 59, 63	15
35, 39, 43, 47	16
3, 7, 11, 15	17
20, 24, 28, 32	20
52, 56, 60, 64	21
4, 8, 12, 16	22
36, 40, 44, 48	23

П1.4 Константы алгоритма MD6

Константы, используемые алгоритмом MD6, приведены в табл. П1.6–П1.7.

Таблица П1.6. Константы Q формирования входной последовательности функции сжатия алгоритма MD6

Номер слова входной последовательности	Значение
0	7311C2812425CFA0
1	6432286434AAC8E7
2	B60450E9EF68B7C1
3	E8FB23908D9F06F1
4	DD2E76CBA691E5BF
5	0CD0D63B2C30BC41
6	1F8CCF6823058F8A
7	54E5ED5B88E3775D
8	4AD12AAE0A6D6031
9	3E7F16BB88222E0D
10	8AF8671D3FB50C2C
11	995AD1178BD25C31
12	C878C1DD04C4B633
13	3B72066C7A1552AC
14	0D6F3522631EFFCB

Константы в табл. П1.6 приведены в шестнадцатеричном виде.

Таблица П1.7. Константы, определяющие количество битов вращения в функции сжатия алгоритма MD6

$(i - 89) \bmod 16$	r_{i-89}	l_{i-89}
0	10	11
1	5	24
2	13	9
3	10	16
4	11	15
5	12	9
6	2	27
7	7	15
8	14	6
9	15	2

Окончание табл. П1.7

$(i - 89) \bmod 16$	r_{i-89}	l_{i-89}
10	7	29
11	13	8
12	11	15
13	7	5
14	6	31
15	12	9

i – номер слова входной последовательности.

П1.5 Константы алгоритмов семейства SHA-2

Раундовые константы алгоритма SHA-256 по порядку от $K_{0,256}$ до $K_{63,256}$ приведены в табл. П1.8 (указаны шестнадцатеричные значения).

Таблица П1.8. Раундовые константы алгоритма SHA-256

428A2F98	71374491	B5C0FBCF	E9B5DBA5
3956C25B	59F111F1	923F82A4	AB1C5ED5
D807AA98	12835B01	243185BE	550C7DC3
72BE5D74	80DEB1FE	9BDC06A7	C19BF174
E49B69C1	EFBE4786	0FC19DC6	240CA1CC
2DE92C6F	4A7484AA	5CB0A9DC	76F988DA
983E5152	A831C66D	B00327C8	BF597FC7
C6E00BF3	D5A79147	06CA6351	14292967
27B70A85	2E1B2138	4D2C6DFC	53380D13
650A7354	766A0ABB	81C2C92E	92722C85
A2BFE8A1	A81A664B	C24B8B70	C76C51A3
D192E819	D6990624	F40E3585	106AA070
19A4C116	1E376C08	2748774C	34B0BCB5
391C0CB3	4ED8AA4A	5B9CCA4F	682E6FF3
748F82EE	78A5636F	84C87814	8CC70208
90BEFFFF	A4506CEB	BEF9A3F7	C67178F2

Раундовые константы алгоритма SHA-512 по порядку от $K_{0,512}$ до $K_{79,512}$ приведены в табл. П1.9 (указаны шестнадцатеричные значения).

Таблица П1.9. Раундовые константы алгоритма SHA-512

428A2F98D728AE22	7137449123EF65CD
B5C0FBCFEC4D3B2F	E9B5DBA58189DBBC
3956C25BF348B538	59F111F1B605D019
923F82A4AF194F9B	AB1C5ED5DA6D8118
D807AA98A3030242	12835B0145706FBE
243185BE4EE4B28C	550C7DC3D5FFB4E2
72BE5D74F27B896F	80DEB1FE3B1696B1
9BDC06A725C71235	C19BF174CF692694
E49B69C19EF14AD2	EFBE4786384F25E3
0FC19DC68B8CD5B5	240CA1CC77AC9C65
2DE92C6F592B0275	4A7484AA6EA6E483
5CB0A9DCBD41FBD4	76F988DA831153B5
983E5152EE66DFAB	A831C66D2DB43210
B00327C898FB213F	BF597FC7BEEF0EE4
C6E00BF33DA88FC2	D5A79147930AA725
06CA6351E003826F	142929670A0E6E70
27B70A8546D22FFC	2E1B21385C26C926
4D2C6DFC5AC42AED	53380D139D95B3DF
650A73548BAF63DE	766A0ABB3C77B2A8
81C2C92E47EDAEE6	92722C851482353B
A2BFE8A14CF10364	A81A664BBC423001
C24B8B70D0F89791	C76C51A30654BE30
D192E819D6EF5218	D69906245565A910
F40E35855771202A	106AA07032BBD1B8
19A4C116B8D2D0C8	1E376C085141AB53
2748774CDF8EEB99	34B0BCB5E19B48A8
391C0CB3C5C95A63	4ED8AA4AE3418ACB
5B9CCA4F7763E373	682E6FF3D6B2B8A3
748F82EE5DEFB2FC	78A5636F43172F60

84C87814A1F0AB72	8CC702081A6439EC
90BEFFFA23631E28	A4506CEBDE82BDE9
BEF9A3F7B2C67915	C67178F2E372532B
CA273ECEEA26619C	D186B8C721C0C207
EADA7DD6CDE0EB1E	F57D4F7FEE6ED178
06F067AA72176FBA	0A637DC5A2C898A6
113F9804BEF90DAE	1B710B35131C471B
28DB77F523047D84	32CAAB7B40C72493
3C9EBE0A15C9BEB	431D67C49C100D4C
4CC5D4BECB3E42B6	597F299CF657E2A
5FCB6FAB3AD6FAEC	6C44198C4A475817

П1.6 Раундовые константы алгоритмов семейства SHA-3

Таблица П1.10. Раундовые константы алгоритмов семейства SHA-3

Номер раунда <i>i</i>	<i>RC[i]</i>
0	00000000000000000001
1	00000000000000008082
2	80000000000000808A
3	8000000080008000
4	00000000000000808B
5	0000000080000001
6	8000000080008081
7	800000000000008009
8	00000000000000008A
9	000000000000000088
10	0000000080008009
11	0000000080000000A
12	000000008000808B
13	80000000000000008B
14	80000000000000008089

Окончание табл. П1.10

Номер раунда i	$RC[i]$
15	800000000000008003
16	800000000000008002
17	80000000000000080
18	00000000000000800A
19	800000008000000A
20	8000000080008081
21	800000000000008080
22	0000000080000001
23	8000000080008008

Константы в табл. П1.10 приведены в шестнадцатеричном виде.

П1.7 Константы алгоритма ГОСТ Р 34.11–2012

Константы, используемые алгоритмом ГОСТ Р 34.11–2012, приведены в табл. П1.11–П1.14.

Таблица П1.11. Таблица замен алгоритма ГОСТ Р 34.11–2012

252	238	221	17	207	110	49	22
251	196	250	218	35	197	4	77
233	119	240	219	147	46	153	186
23	54	241	187	20	205	95	193
249	24	101	90	226	92	239	33
129	28	60	66	139	1	142	79
5	132	2	174	227	106	143	160
6	11	237	152	127	212	211	31
235	52	44	81	234	200	72	171
242	42	104	162	253	58	206	204
181	112	14	86	8	12	118	18
191	114	19	71	156	183	93	135
21	161	150	41	16	123	154	199
243	145	120	111	157	158	178	177

Окончание табл. П1.11

50	117	25	61	255	53	138	126
109	84	198	128	195	189	13	87
223	245	36	169	62	168	67	201
215	121	214	246	124	34	185	3
224	15	236	222	122	148	176	188
220	232	40	80	78	51	10	74
167	151	96	115	30	0	98	68
26	184	56	130	100	159	38	65
173	69	70	146	39	94	85	47
140	163	165	125	105	213	149	59
7	88	179	64	134	172	29	247
48	55	107	228	136	217	231	137
225	27	131	73	76	63	248	254
141	83	170	144	202	216	133	97
32	113	103	164	45	43	9	91
203	155	37	208	190	229	108	82
89	166	116	210	230	244	180	192
209	102	175	194	57	75	99	182

В i -й ячейке табл. П1.11 содержится выходное значение замены входного байта со значением i .

Таблица П1.12. Таблица байтовой перестановки алгоритма ГОСТ Р 34.11–2012

0	8	16	24	32	40	48	56	1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58	3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60	5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62	7	15	23	31	39	47	55	63

i -я ячейка табл. П1.12 содержит новую позицию i -го входного байта.

Таблица П1.13. Матрица для умножения в алгоритме ГОСТ Р 34.11–2012

8E20FAA72BA0B470	47107DDD9B505A38
AD08B0E0C3282D1C	D8045870EF14980E
6C022C38F90A4C07	3601161CF205268D
1B8E0B0E798C13C8	83478B07B2468764
A011D380818E8F40	5086E740CE47C920
2843FD2067ADEA10	14AFF010BDD87508
0AD97808D06CB404	05E23C0468365A02
8C711E02341B2D01	46B60F011A83988E
90DAB52A387AE76F	486DD4151C3DFDB9
24B86A840E90F0D2	125C354207487869
092E94218D243CBA	8A174A9EC8121E5D
4585254F64090FA0	ACCC9CA9328A8950
9D4DF05D5F661451	C0A878A0A1330AA6
60543C50DE970553	302A1E286FC58CA7
18150F14B9EC46DD	0C84890AD27623E0
0642CA05693B9F70	0321658CBA93C138
86275DF09CE8AAA8	439DA0784E745554
AFC0503C273AA42A	D960281E9D1D5215
E230140FC0802984	71180A8960409A42
B60C05CA30204D21	5B068C651810A89E
456C34887A3805B9	AC361A443D1C8CD2
561B0D22900E4669	2B838811480723BA
9BCF4486248D9F5D	C3E9224312C8C1A0
EFFA11AF0964EE50	F97D86D98A327728
E4FA2054A80B329C	727D102A548B194E
39B008152ACB8227	9258048415EB419D
492C024284FBAEC0	AA16012142F35760
550B8E9E21F7A530	A48B474F9EF5DC18
70A6A56E2440598E	3853DC371220A247
1CA76E95091051AD	0EDD37C48A08A6D8
07E095624504536C	8D70C431AC02A736
C83862965601DD1B	641C314B2B8EE083

Таблица П1.14. Раундовые константы процедуры вычисления псевдоключей алгоритма ГОСТ Р 34.11–2012

C_i	Шестнадцатеричное значение
C_1	b1085bda1ecadae9ebcb2f81c0657c1f2f6a76432e45d016714eb88d7585c4fc4b7ce09 192676901a2422a08a460d31505767436cc744d23dd806559f2a64507
C_2	6fa3b58aa99d2f1a4fe39d460f70b5d7f3fea720a232b9861d55e0f16b501319ab5176 b12d699585cb561c2db0aa7ca55dda21bd7cbcd56e679047021b19bb7
C_3	f574dcac2bce2fc70a39fc286a3d843506f15e5f529c1f8bf2ea7514b1297b7bd3e20fe4 90359eb1c1c93a376062db09c2b6f443867adb31991e96f50aba0ab2
C_4	ef1fdfb3e81566d2f948e1a05d71e4dd488e857e335c3c7d9d721cad685e353fa9d72c8 2ed03d675d8b71333935203be3453eaa193e837f1220cbebc84e3d12e
C_5	4bea6bacad4747999a3f410c6ca923637f151c1f1686104a359e35d7800ffffbdbfcfd174 7253af5a3dff00b723271a167a56a27ea9ea63f5601758fd7c6cf57
C_6	ae4faeae1d3ad3d96fa4c33b7a3039c02d66c4f95142a46c187f9ab49af08ec6cffaa6b7 1c9ab7b40af21f66c2bec6b6bf71c57236904f35fa68407a46647d6e
C_7	f4c70e16eeaac5ec51ac86feb240954399ec6c7e6bf87c9d3473e33197a93c90992abc5 2d822c3706476983284a05043517454ca23c4af38886564d3a14d493
C_8	9b1f5b424d93c9a703e7aa020c6e41414eb7f8719c36de1e89b443b4ddbc49af4892b cb929b069069d18d2bd1a5c42f36acc2355951a8d9a47f0dd4bf02e71e
C_9	378f5a541631229b944c9ad8ec165fde3a7d3a1b258942243cd955b7e00d0984800a4 40bdbb2ceb17b2b8a9aa6079c540e38dc92cb1f2a607261445183235adb
C_{10}	abbedea680056f52382ae548b2e4f3f38941e71cff8a78db1fffe18a1b3361039fe76702 af69334b7a1e6c303b7652f43698fad1153bb6c374b4c7fb98459ced
C_{11}	7bcd9ed0efc889fb3002c6cd635afe94d8fa6bbbebab076120018021148466798a1d71 efea48b9caefbacd1d7d476e98dea2594ac06fd85d6bcaa4cd81f32d1b
C_{12}	378ee767f11631bad21380b00449b17acda43c32bcd1d77f82012d430219f9b5d80ef 9d1891cc86e71da4aa88e12852faf417d5d9b21b9948bc924af11bd720

Список сокращений

ABFT	Asynchronous Byzantine Fault Tolerance – асинхронная задача византийских генералов
AES	Advanced Encryption Standard – улучшенный стандарт шифрования
AGPL	Attero General Public License (тип лицензии на свободное программное обеспечение)
API	Application Programming Interface – интерфейс разработки приложений
ASCII	American Standard Code for Information Interchange – американский стандартный код для обмена информацией
ASIC	Application-Specific Integrated Circuit – интегральная схема специального назначения
BFT	Byzantine Fault Tolerance (консенсус на основе задачи византийских генералов)
BIP	Bitcoin Improvement Proposal – предложение по улучшению системы Биткойн
BLS	Boneh-Lynn-Shacham (алгоритм электронной подписи)
BTC	Bitcoin – биткойн (единица криптовалюты)
CRL	Certificate Revocation List – список отозванных сертификатов
CVE	Common Vulnerabilities and Exposures (база данных общезвестных уязвимостей информационной безопасности)
DAG	Directed Acyclic Graph – направленный ациклический граф
DBFT	Delegated Byzantine Fault Tolerance – делегированная задача византийских генералов
DES	Data Encryption Standard – стандарт шифрования данных
DLP	Discrete Logarithm Problem – задача дискретного логарифмирования (в мультиплексивной группе конечного поля)
DLT	Distributed Ledger Technologies – технологии распределенного реестра
DoS	Denial of Service – отказ в обслуживании

DPoS	Delegated Proof of Stake – делегированное доказательство доли владения
DSA	Digital Signature Algorithm – алгоритм цифровой подписи
DSS	Digital Signature Standard – стандарт цифровой подписи
eBACS	ECRYPT Benchmarking of Cryptographic Systems (сравнительный анализ крипtosистем в рамках проекта ECRYPT)
eBASH	ECRYPT Benchmarking of All Submitted Hashes (анализ производительности хеш-функций в рамках европейского криптографического проекта ECRYPT)
ECDLP	Elliptic Curve Discrete Logarithm Problem – задача нахождения дискретного логарифма в группе точек эллиптической кривой
ECDSA	Elliptic Curve Digital Signature Algorithm – алгоритм электронной подписи на эллиптических кривых
EdDSA	Edwards-curve Digital Signature Algorithm – алгоритм электронной подписи на основе кривых Эдвардса
ERC	Ethereum Request for Comments (семейство информационных документов платформы Эфириум)
EVM	Ethereum Virtual Machine – виртуальная машина Эфириум
FBA	Federated Byzantine Agreement – федеративное византийское соглашение
FIPS	Federal Information Processing Standard (семейство стандартов обработки информации США)
GF	Galois Field (конечное поле)
GHOST	Greedy Heaviest Observed Subtree (алгоритм определения корректной цепочки блоков)
GNU	GNU is Not Unix (проект по разработке свободного программного обеспечения)
HD	Hierarchical Deterministic Protocol – иерархический детерминированный протокол
HMAC	Hash-based Message Authentication Code – код аутентификации сообщения на основе хеширования
IBC	Integrated Broadband Communication – интегрированная широкополосная связь
IBM	International Business Machines (известная компания – производитель аппаратного и программного обеспечения)

ID	Identifier – идентификатор
IDE	Integrated Development Environment – интегрированная среда разработки
Idtx	Transaction Identifier – идентификатор транзакции
IEEE	Institute of Electrical and Electronics Engineers – Институт инженеров электротехники и электроники
IP	Internet Protocol – межсетевой протокол
ISO	International Organization for Standardization – Международная организация по стандартизации
IV	Initialization Vector – вектор инициализации
JSON	JavaScript Object Notation (текстовый формат обмена данными)
LIB	Last Irreversible Block – последний необратимый блок
LPoS	Leased Proof of Stake – арендованное доказательство доли владения
MAC	Message Authentication Code – код аутентификации сообщения
MASF	Miner Activated Soft Fork – активируемый майнером soft fork
MD	Message Digest (семейство алгоритмов хеширования)
MIT	Massachusetts Institute of Technology – Массачусетский технологический институт
MMO	Matyas-Meyer-Oseas – алгоритм Матиаса–Мейера–Осиаса
MOV	Menezes-Okamoto-Vanstone (атака на криптографические алгоритмы, основанные на эллиптических кривых)
NFS	Number Field Sieve – метод решета числового поля (метод нахождения дискретного логарифма в конечных полях и факторизации натуральных чисел)
NIST	National Institute of Standards and Technology (Национальный институт стандартов и технологий США)
NMAC	Nested Message Authentication Code – вложенный код аутентификации сообщения
NPM	Node Package Manager (менеджер пакетов, входящий в состав Node.js)
OCSP	Online Certificate Status Protocol – протокол определения статуса сертификата

P2PKH	Pay-To-Public-Key-Hash – «плата за хеш открытого ключа» (вид транзакции в системе Биткойн)
P2SH	Pay-To-Script-Hash – «плата за хеш скрипта» (вид транзакции в системе Биткойн)
P2WPKH	Pay-To-Witness-Public-Key-Hash – «плата за свидетельство хеша открытого ключа» (вид транзакции в системе Биткойн)
P2WSH	Pay-To-Witness-Script-Hash – «плата за свидетельство хеша скрипта» (вид транзакции в системе Биткойн)
PBFT	Practical Byzantine Fault Tolerance – практическая задача византийских генералов
PBKDF	Password-Based Key Derivation Function – функция получения ключа на основе пароля
PEM	Privacy-Enhanced Mail (стандарт защищенной электронной почты)
PKI	Public Key Infrastructure – инфраструктура открытых ключей
PoA	Proof of Activity (консенсус на основе доказательства деятельности)
PoET	Proof of Elapsed Time (консенсус на основе доказательства потраченного времени)
PoS	Proof of Stake (консенсус на основе доказательства владения долей)
PoST	Proof of Stake Time (консенсус на основе доказательства времени ставки)
PoW	Proof of Work (консенсус на основе доказательства работы)
PoWeight	Proof of Weight (консенсус на основе доказательства веса)
QR	Quick Response (тип машиносчитываемого двумерного штрих-кода)
RACE	Research and Development in Advanced Communication Technologies in Europe (европейский проект по исследованиям и разработкам улучшенных технологий связи)
RBFT	Redundant Byzantine Fault Tolerance – задача византийских генералов с избыточностью
RFC	Request for Comments (семейство информационных документов сети Интернет)
RFU	Reserved for Future Usage (зарезервированное поле данных)

RIPE	RACE Integrity Primitive Evaluation (оценка примитивов конт- роля целостности в проекте RACE)
RIPEMD	RIPE Message Digest (дайджест сообщения в проекте RIPE)
RSA	Rivest-Shamir-Adleman (асимметричная криптосистема)
SDK	Software Development Kit – комплект для разработки про- граммного обеспечения
SEA	Schoof-Elkies-Atkin (алгоритм подсчета количества точек эл- липтической кривой над конечным полем)
SHA	Secure Hash Algorithm – алгоритм безопасного хеширования
SHS	Secure Hash Standard – стандарт безопасного хеширования
SPA	Simple Power Analysis (атака на основе анализа потребляемой мощности)
SQL	Structured Query Language – язык структурированных запросов
TAPoS	Transaction As Proof of Stake – транзакция как доказательство доли
TEA	Tiny Encryption Algorithm – «миниатюрный алгоритм шифрования»
UASF	User Activated Soft Fork – активируемый пользователем softfork
URI	Uniform Resource Identifier – унифицированный идентификатор ресурса
URL	Uniform Resource Locator – унифицированный адрес ресурса
USB	Universal Serial Bus – универсальная последовательная шина
UTXO	Unspent Transaction Output – неизрасходованный выход транз- акции
WASM	WebAssembly (формат и технология выполнения байт-кода)
WIF	Wallet Import Format – формат импорта кошелька
WSL	Windows Subsystem for Linux – подсистема Windows для Linux
XEdDSA	Extended Edwards-curve Digital Signature Algorithm – расширен- ный алгоритм электронной подписи на основе кривых Эдвардса
XOR	Exclusive-OR (логическая операция «исключающее или»)
YAC	Yet Another Consensus – «еще один консенсус»
zk-SNARK	Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (семейство криптографических протоколов на основе доказа- тельства с нулевым разглашением)

ГОСТ	Государственный стандарт (семейство национальных стандартов РФ и межгосударственных стандартов)
ОАО	Открытое акционерное общество
ОС	Операционная система
ПИН	Персональный идентификационный номер
ПО	Программное обеспечение
РЖД	Российские железные дороги
РФ	Российская Федерация
СКЗИ	Средство криптографической защиты информации
СКСО	Смарт-контракты сервисного обслуживания
США	Соединенные Штаты Америки
ТК	Технический комитет
ФГИС	Федеральная государственная информационная система
ФСБ	Федеральная служба безопасности
ЭП	Электронная подпись

Перечень рисунков

Рисунок 1.1. Схема Меркля–Дамгорда	10
Рисунок 1.2. Фазы преобразований криптографической губки.....	11
Рисунок 1.3. Схема алгоритма ММО.....	12
Рисунок 1.4. Схема алгоритма Милягучи–Пренеля	13
Рисунок 1.5. Схема использования MAC	14
Рисунок 1.6. Вычисление HMAC.....	16
Рисунок 1.7. Вычисление NMAC.....	17
Рисунок 1.8. Дерево Меркля.....	17
Рисунок 1.9. Таблица паролей и хеш-кодов	23
Рисунок 1.10. Таблица цепочек хеш-кодов	24
Рисунок 1.11. Пример коллизии в цепочке хеш-кодов	24
Рисунок 1.12. Таблица цепочек переменной длины	25
Рисунок 1.13. Радужная таблица.....	26
Рисунок 1.14. Параллельный поиск коллизий	29
Рисунок 1.15. Мультиколлизия	30
Рисунок 1.16. Итерация алгоритма SHI1	31
Рисунок 1.17. Распространение и коррекция разности в бите № 1 <i>i</i> -х фрагментов расширенных блоков алгоритма SHI1	33
Рисунок 1.18. Пример структуры алгоритма, подверженного атаке «встреча посередине».....	35
Рисунок 1.19. Атака «встреча посередине»	36
Рисунок 1.20. Копирование блока данных в состояние алгоритма MD2	40
Рисунок 1.21. Итерация алгоритма MD4	44
Рисунок 1.22. Итерация алгоритма MD5	47
Рисунок 1.23. Структура алгоритма MD6	50
Рисунок 1.24. Входная последовательность функции сжатия алгоритма MD6....	51
Рисунок 1.25. Идентификатор экземпляра функции сжатия <i>U</i> в алгоритме MD6.....	51
Рисунок 1.26. Дополнительное слово <i>V</i> входной последовательности функции сжатия алгоритма MD6.....	52
Рисунок 1.27. Итерация функции сжатия алгоритма MD6.....	53
Рисунок 1.28. Функция сжатия алгоритма MD6.....	53
Рисунок 1.29. Последовательный вариант алгоритма MD6	54
Рисунок 1.30. Входные данные функции сжатия при последовательном вычислении хеш-кода алгоритмом MD6	55

Рисунок 1.31. Структура алгоритма MD6 при $L = 1$	56
Рисунок 1.32. Ключ как часть входной последовательности функции сжатия алгоритма MD6.....	57
Рисунок 1.33. Итерация алгоритма RIPEMD-160	64
Рисунок 1.34. Схема обработки блока входных данных в алгоритме RIPEMD-160	65
Рисунок 1.35. Итерация алгоритма RIPEMD-128	67
Рисунок 1.36. Схема обработки блока входных данных в алгоритме RIPEMD-256	69
Рисунок 1.37. Расширение обрабатываемого блока в алгоритме SHA.....	71
Рисунок 1.38. Итерация алгоритма SHA.....	71
Рисунок 1.39. Расширение обрабатываемого блока в алгоритме SHA-1	73
Рисунок 1.40. Расширение обрабатываемого блока в алгоритме SHA-256	74
Рисунок 1.41. Итерация алгоритма SHA-256	75
Рисунок 1.42. Расширение обрабатываемого блока в алгоритме SHA-512	77
Рисунок 1.43. Итерация алгоритма SHA-512	78
Рисунок 1.44. Схематичное изображение раунда функции $f()$ алгоритма SHA-3	82
Рисунок 1.45. Конструкция Миягучи–Пренеля в алгоритме ГОСТ Р 34.11–2012	89
Рисунок 1.46. Структура внутреннего блочного шифра алгоритма ГОСТ Р 34.11–2012	89
Рисунок 2.1. Геометрическое представление суммы точек эллиптической кривой.....	97
Рисунок 2.2. Геометрическое представление удвоения точки эллиптической кривой	98
Рисунок 3.1. Общий вид транзакции.....	123
Рисунок 3.2. Пример структуры криптовалютной транзакции.....	124
Рисунок 3.3. Криптовалютная транзакция, содержащая только выходы	124
Рисунок 3.4. Формирование и проверка подписи транзакции	126
Рисунок 3.5. Количество транзакций в мемпуле (вверху) и график стоимости биткойна (внизу)	129
Рисунок 3.6. Дерево Меркля для шести транзакций в блоке	130
Рисунок 3.7. Дерево Меркля для пяти транзакций в блоке.....	131
Рисунок 3.8. Пример построения цепочки блоков	140
Рисунок 3.9. Одиннадцатый блок системы Биткойн	141
Рисунок 3.10. Блок системы Биткойн № 717 000.....	142
Рисунок 3.11. Пример форка для блокчейн-цепочки.....	143
Рисунок 3.12. Аппаратный кошелек Ledger Nano X	152
Рисунок 4.1. Упрощенное представление цепочки блоков биткойна.....	154

Рисунок 4.2. Пример связей между поступлениями и тратами для нескольких транзакций	155
Рисунок 4.3. Первый блок биткойн-цепочки	157
Рисунок 4.4. Примеры форков	158
Рисунок 4.5. Транзакции в блоке биткойна с высотой 100	160
Рисунок 4.6. Транзакции в блоке биткойна с высотой 200 000	160
Рисунок 4.7. Данные блока биткойна с высотой 599 999	161
Рисунок 4.8. Пример коллизии для дерева Меркля	162
Рисунок 4.9. Coinbase-транзакция в блоке биткойна с высотой 10	163
Рисунок 4.10. Coinbase-транзакция в блоке биткойна с высотой 300 000	164
Рисунок 4.11. Связь входов и выходов в транзакциях	164
Рисунок 4.12. Входы и выходы транзакции в блоке биткойна с высотой 718 300	165
Рисунок 4.13. Выходы транзакции в блоке биткойна с высотой 718 300	165
Рисунок 4.14. Преобразование выхода во вход	169
Рисунок 4.15. Создание P2PKH для получения платежа	170
Рисунок 4.16. Разблокирование выхода P2PKH для следующей траты	171
Рисунок 4.17. Схема последовательной проверки всех элементов	172
Рисунок 4.18. Пример преобразования выхода во вход по принципу P2PKH	174
Рисунок 4.19. Создание P2SH для получения платежа	175
Рисунок 4.20. Разблокирование выхода P2SH для следующей траты	175
Рисунок 4.21. Пример преобразования выхода во вход по принципу P2SH	177
Рисунок 4.22. Пример преобразования выхода во вход по принципу мультиподписи	178
Рисунок 4.23. Пример транзакции по принципу «Только открытый ключ»	179
Рисунок 4.24. Пример транзакции с нулевыми выходами (заморозка биткойнов)	181
Рисунок 4.25. Пример транзакции с нулевыми выходами (любовное послание)	182
Рисунок 4.26. Пример нестандартной транзакции	184
Рисунок 4.27. Кошелек s-272edf45031dd498e7b3ae89e11ff21b	184
Рисунок 4.28. Выписка по кошельку s-272edf45031dd498e7b3ae89e11ff21b	185
Рисунок 4.29. Дублирующая транзакция для блоков 91812 и 91842	186
Рисунок 4.30. Дублирующая транзакция для блоков 91722 и 91880	186
Рисунок 4.31. Транзакции по первому типу блокировки	189
Рисунок 4.32. Транзакции по второму типу блокировки	189
Рисунок 4.33. Комиссия для блока 715 000	190
Рисунок 4.34. Транзакции с различной стоимостью комиссии в майнинг-пуле	191

Рисунок 4.35. Пример транзакции с выходом сдачи.....	192
Рисунок 4.36. Пример перехода выхода во вход для транзакции типа P2WPKH	194
Рисунок 4.37. Пример перехода выхода во вход для транзакции типа P2WSH	196
Рисунок 4.38. Пример перехода выхода во вход для транзакции типа P2WSH с обратной совместимостью	197
Рисунок 4.39. Схема взаимодействия Алисы и Боба при формировании канала микроплатежей.....	200
Рисунок 4.40. Пример транзакции объединения монет	202
Рисунок 4.41. Большая анонимная транзакция CoinJoin	203
Рисунок 4.42. Преобразование случайной последовательности в мнемоническую фразу	205
Рисунок 4.43. Расширение ключа с использованием мнемонической фразы.....	206
Рисунок 4.44. Вычисление дочерних ключей	208
Рисунок 4.45. Создание мастер-ключей.....	208
Рисунок 4.46. Глобальное состояние сети Эфириум.....	210
Рисунок 4.47. Установка криптовалютного кошелька MetaMask.....	219
Рисунок 4.48. Выбор секретной фразы кошелька MetaMask	220
Рисунок 4.49. Получение эфира в тестовой сети	221
Рисунок 4.50. Просмотр информации о тестовых средствах	221
Рисунок 4.51. Создание нового счета	222
Рисунок 4.52. Перевод средств между двумя счетами	223
Рисунок 4.53. Главное окно Ganache	223
Рисунок 4.54. Вкладка «Server»	224
Рисунок 4.55. Вкладка «Accounts and Keys».....	225
Рисунок 4.56. Вкладка «Chain»	226
Рисунок 4.57. Созданные аккаунты тестовой сети	226
Рисунок 4.58. Информация о блоке	227
Рисунок 4.59. Информация о закрытом ключе.....	227
Рисунок 4.60. Загрузка в MetaMask данных сети Ganache	228
Рисунок 4.61. Создание нового аккаунта с помощью Geth	229
Рисунок 4.62. Файл кошелька.....	229
Рисунок 4.63. Файл генезис-блока.....	230
Рисунок 4.64. Инициализация генезис-блока	230
Рисунок 4.65. Инициализация цепочки	231
Рисунок 4.66. Команды модуля eth.....	232
Рисунок 4.67. Просмотр аккаунтов в сети	232

Рисунок 4.68. Просмотр баланса аккаунта по индексу	233
Рисунок 4.69. Просмотр баланса аккаунта с помощью дополнительной переменной.....	233
Рисунок 4.70. Команды модуля personal	233
Рисунок 4.71. Создание нового аккаунта.....	234
Рисунок 4.72. Основная структура контракта на языке Solidity.....	236
Рисунок 4.73. Обращение к элементам структуры.....	238
Рисунок 4.74. Структура функции	238
Рисунок 4.75. Тестовый контракт	240
Рисунок 4.76. Выбор виртуальной машины.....	241
Рисунок 4.77. Функция создания транзакции.....	251
Рисунок 4.78. Функция получения транзакции по идентификатору.....	252
Рисунок 4.79. Функция обновления транзакции.....	252
Рисунок 4.80. Параметры для подключения к блокчейн-сети	252
Рисунок 4.81. Вызов функции создания транзакции	253
Рисунок 4.82. Вызов функции получения транзакции по идентификатору	253

Перечень таблиц

Таблица 1.1. Различающиеся биты фрагментов расширенных блоков при дифференциальном криптоанализе алгоритма SH1.....	33
Таблица 1.2. Быстродействие ряда известных алгоритмов хеширования	40
Таблица 1.3. Начальное заполнение регистров алгоритма MD4	42
Таблица 1.4. Раундовые функции алгоритма MD4	43
Таблица 1.5. Модифицирующие константы алгоритма MD4	43
Таблица 1.6. Количество битов сдвига в зависимости от номера итерации алгоритма MD4	43
Таблица 1.7. Начальное заполнение регистров в расширенном варианте алгоритма MD4	44
Таблица 1.8. Модифицирующие константы в расширенном варианте алгоритма MD4	45
Таблица 1.9. Начальное заполнение регистров алгоритма MD5	46
Таблица 1.10. Раундовые функции алгоритма MD5	47
Таблица 1.11. Позиции обратной связи в итерации алгоритма MD6	52
Таблица 1.12. Начальное заполнение регистров алгоритма RIPEMD-160	60
Таблица 1.13. Раундовые функции первого потока преобразований алгоритма RIPEMD-160	61
Таблица 1.14. Константы выбора слов блока входных данных первого потока преобразований алгоритма RIPEMD-160	62
Таблица 1.15. Раундовые константы первого потока преобразований алгоритма RIPEMD-160	62
Таблица 1.16. Константы, определяющие количество битов вращения первого потока преобразований алгоритма RIPEMD-160	62
Таблица 1.17. Раундовые функции второго потока преобразований алгоритма RIPEMD-160	63
Таблица 1.18. Константы выбора слов блока входных данных второго потока преобразований алгоритма RIPEMD-160	63
Таблица 1.19. Раундовые константы второго потока преобразований алгоритма RIPEMD-160	63
Таблица 1.20. Константы, определяющие количество битов вращения второго потока преобразований алгоритма RIPEMD-160	64
Таблица 1.21. Вычисление раундовых констант алгоритма RIPEMD-160	66
Таблица 1.22. Перестановка слов блока входных данных в алгоритме RIPEMD-160	66
Таблица 1.23. Порядок применения перестановок слов блока входных данных в алгоритме RIPEMD-160.....	66

Таблица 1.24. Раундовые функции алгоритма RIPEMD-128	68
Таблица 1.25. Вычисление раундовых констант алгоритма RIPEMD-128	68
Таблица 1.26. Начальное заполнение регистров алгоритма SHA.....	70
Таблица 1.27. Модифицирующие константы алгоритма SHA.....	72
Таблица 1.28. Функции итераций алгоритма SHA.....	72
Таблица 1.29. Начальное заполнение регистров алгоритма SHA-256.....	74
Таблица 1.30. Начальное заполнение регистров алгоритма SHA-512.....	76
Таблица 1.31. Начальное заполнение регистров алгоритма SHA-384.....	78
Таблица 1.32. Начальное заполнение регистров алгоритма SHA-224.....	79
Таблица 3.1. Поиск заданного хеш-кода	132
Таблица 3.2. Поиск хеш-кодов разной сложности	133
Таблица 4.1. Соответствие доли биткойна и количества сатоши	153
Таблица 4.2. Основные команды языка Script	166
Таблица 4.3. Основные скрипты сценария «Плата за хеш открытого ключа» ...	172
Таблица 4.4. Основные скрипты сценария «Плата за хеш скрипта».....	176
Таблица 4.5. Основные скрипты сценария «Коллективная подпись».....	178
Таблица 4.6. Альтернативное представление сценария «Коллективная подпись»	178
Таблица 4.7. Основные скрипты сценария «Только открытый ключ»	179
Таблица 4.8. Основные скрипты сценария «Транзакция с нулевыми данными»	180
Таблица 4.9. Основные виды нестандартных транзакций	183
Таблица 4.10. Основные поля транзакции P2WPKH.....	194
Таблица 4.11. Основные поля транзакции P2WSH	195
Таблица 4.12. Транзакция P2WSH с обратной совместимостью	196
Таблица 4.13. Пример задания исходных данных системы.....	237
Таблица П1.1. Таблица замен алгоритма MD2.....	261
Таблица П1.2. Индексы используемых слов блока сообщения в зависимости от номера итерации алгоритма MD4	262
Таблица П1.3. Индексы используемых слов блока сообщения в зависимости от номера итерации алгоритма MD5	263
Таблица П1.4. Модифицирующие константы в зависимости от номера итерации алгоритма MD5.....	264
Таблица П1.5. Количество битов циклического сдвига влево в зависимости от номера итерации алгоритма MD5	266
Таблица П1.6. Константы Q формирования входной последовательности функции сжатия алгоритма MD6.....	267
Таблица П1.7. Константы, определяющие количество битов вращения в функции сжатия алгоритма MD6	267
Таблица П1.8. Раундовые константы алгоритма SHA-256	268

Таблица П1.9. Раундовые константы алгоритма SHA-512	269
Таблица П1.10. Раундовые константы алгоритмов семейства SHA-3.....	270
Таблица П1.11. Таблица замен алгоритма ГОСТ Р 34.11–2012	271
Таблица П1.12. Таблица байтовой перестановки алгоритма ГОСТ Р 34.11–2012	272
Таблица П1.13. Матрица для умножения в алгоритме ГОСТ Р 34.11–2012	273
Таблица П1.14. Раундовые константы процедуры вычисления псевдоключей алгоритма ГОСТ Р 34.11–2012.....	274

Перечень источников

- [1] Адрес криптовалюты (Ethereum, Litecoin, Ripple, Dogecoin) – что это, как выглядит, где найти и взять. Полный обзор с примерами и пояснениями // <https://profinvestment.com/address-crypto-currency/>.
- [2] Адрес s-272edf45031dd498e7b3ae89e11ff21b // <https://blockchair.com/ru/bitcoin/address/s-272edf45031dd498e7b3ae89e11ff21b>.
- [3] Алгоритм Ethash // <https://habr.com/ru/post/534754/>.
- [4] Асмаков А. С помощью CoinJoin совершена самая крупная анонимная транзакция в истории биткойна // <https://forklog.com/s-pomoshhyu-coinjoin-sovershena-samaya-krupnaya-anonimnaya-tranzaktsiya-v-istorii-bitkoina/>.
- [5] Блок биткойна № 10 // <https://www.blockchain.com/btc/block/10>.
- [6] Блок биткойна № 717 000 // <https://www.blockchain.com/btc/block/717000>.
- [7] В сети Ethereum состоялся хардфорк Istanbul. На очереди – Berlin // <https://bloomchain.ru/newsfeed/v-seti-ethereum-sostoyalsya-hardfork-istanbul-na-ochederi-berlin/>.
- [8] Валидатор в криптовалюте // <https://finswin.com/kripto/terminologiya/validator.html>.
- [9] Василенко О. Н. Теоретико-числовые алгоритмы в криптографии. М.: МЦНМО, 2003. 328 с.
- [10] Гибридная блокчейн-платформа для бизнеса и государства // <https://wavesenterprise.com/ru> – Waves Enterprise.
- [11] ГОСТ 28147–89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования.
- [12] ГОСТ Р 34.10–2001. Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи.
- [13] ГОСТ Р 34.10–2012. Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи.
- [14] ГОСТ Р 34.11–94. Информационная технология. Криптографическая защита информации. Функция хэширования¹.
- [15] ГОСТ Р 34.11–2012. Информационная технология. Криптографическая защита информации. Функция хэширования.
- [16] Деревья Меркля в Эфириуме // <https://imgun.wordpress.com/2015/11/22/merkling-in-ethereum/>.
- [17] Джесс Л. Куда пропали 182 биткойна из «формулы Сатоши»? // <https://forklog.com/kuda-propali-182-bitkoina-iz-formuly-satoshi>.
- [18] Дистанционное электронное голосование на блокчейне // <https://we.vote> – Waves Enterprise.

¹ Написание слова здесь, а также далее в источниках [15] и [29] дано в соответствии с оригиналом.

- [19] Запущена новая тестовая сеть Эфириума Kovan на замену пострадавшей от спам-атак Ropsten//<https://bits.media/zapushchena-novaya-testovaya-set-efiriuma-kovan-na-zamenu-postradavshay-ot-spam-atak-ropsten/>.
- [20] Захватывающая история The DAO: работа над ошибками // <https://forklog.com/zahvatyvayushhaya-istoriya-the-dao-rabota-nad-oshibkami/>.
- [21] Как устроен Ethereum и смарт-контракты // <https://vas3k.ru/blog/ethereum/>.
- [22] Какое количество биткойнов утеряно навсегда в данный момент? // <https://cryptonews.net/ru/news/bitcoin/244807/> – 21 ноября 2019 г.
- [23] КриптоВече. Система онлайн-голосования на блокчейне // <https://cryptoveche.dltc.spbu.ru> – Санкт-Петербургский государственный университет.
- [24] Крупнейший биткойн-пул пообещал не осуществлять атаку 51 % // <https://xakep.ru/2014/01/11/61859/>.
- [25] *Лидл Р., Нидеррайтер Г. Н.* Конечные поля: в 2 т. / пер. с англ. М.: Мир, 1988. 430 с.
- [26] *Маршалко Г.* Национальная и международная стандартизация российских криптографических алгоритмов // PC Week/RE. 2014. 2 дек. № 21 (876).
- [27] Материалы Технического комитета по стандартизации «Криптографическая защита информации» (ТК 26) // <http://www.tc26.ru>.
- [28] *Матюхин Д. В.* Об асимптотической сложности дискретного логарифмирования в поле $GF(p)$ // Дискретная математика. 2003. Т. 15. № 1. С. 28–49.
- [29] *Матюхин Д. В., Рудской В. И., Шишкин В. А.* Перспективный алгоритм хэширования // https://www.ruscrypto.ru/resource/archive/rc2010/files/02_matyukhin_rudskoy_shishkin.pdf – презентация доклада на конференции РусКрипто'2010 – 2 апреля 2010.
- [30] *Накамото С.* Биткойн: система цифровой пищевой наличности // https://bitcoin.org/files/bitcoin-paper/bitcoin_ru.pdf.
- [31] Обзор криптографического алгоритма Ethash, майнинг криптовалют // <https://crypta.guru/kriptoanalyty/algorithm-ethash/>.
- [32] Общее количество неподтвержденных транзакций в мемпуле // <https://www.blockchain.com/charts/mempool-count>.
- [33] Онлайн-калькулятор unix time stamp // <https://www.bl2.ru/programing/timestamp.html>.
- [34] *Панасенко С. П.* Алгоритмы шифрования. Специальный справочник. СПб.: БХВ-Петербург, 2009. 576 с.
- [35] *Панасенко С. П.* Обзор атак на приложения и протоколы, использующие алгоритм MD5, части 1–9 // <http://www.panasenko.ru> – 2013.
- [36] Платформа Мастерчейн // <https://www.dltru.org/platforma-mastercheyn> – ООО «Системы распределенного реестра».
- [37] Подробно об обновлении Segregated Witness и последствиях его принятия в Bitcoin // <https://habr.com/ru/company/distributedlab/blog/418853/> – 31 июля 2018 г.
- [38] *Применко Э. А.* Алгебраические основы криптографии. М.: URSS, 2013. 288 с.
- [39] Проект Ethereum 2.0 вступил в нулевую фазу – курс ETH обвалился на 8 % // <https://bloomchain.ru/newsfeed/proekt-ethereum-2-0-vstupil-v-nulevuju-fazu-kurs-eth-obvalilsja-na-8>.

- [40] ПроКСи: смарт-контракты для бизнеса и госорганизаций // <https://procsy.ru/#solutions> – АО «Промышленные Криптоустановки».
- [41] Развитие технологии распределенного реестра // <https://www.fintechru.org/directions/raspredelenny-reestr/> – Ассоциация ФИНТЕХ.
- [42] Разработчики Ethereum представили проект спецификаций хардфорка Berlin//<https://bloomchain.ru/newsfeed/razrabotchiki-ethereum-predstavili-proekt-spetsifikatsii-hardforka-berlin>.
- [43] Рекомендации по стандартизации Р 1323565.1.024–2019. Информационная технология. Криптографическая защита информации. Параметры эллиптических кривых для криптографических алгоритмов и протоколов.
- [44] Романец Ю. В., Тимофеев П. А., Шаньгин В. Ф. Защита информации в компьютерных системах и сетях. 2-е изд. М.: Радио и связь, 2001. 376 с.
- [45] Семаев И. А. О вычислении логарифмов на эллиптических кривых // Дискретная математика. 1996. Т. 8. № 1. С. 65–71.
- [46] Смарт Н. Криптография / пер. с англ. М.: Техносфера, 2006. 528 с.
- [47] Средняя рыночная цена в USD на основных биржах биткойнов // <https://www.blockchain.com/charts/market-price>.
- [48] Стандарт PBKDF2 // <https://defcon.ru/cryptography/485/> – 16.06.2015.
- [49] Форк Эфириума (Ethereum) // <https://coinnet.ru/fork-ethereum/>.
- [50] Что такое газ в Эфириуме? Сколько платить за транзакции Ethereum // <https://2bitcoins.ru/cto-takoe-gas-v-ethereum-skolko-platit-za-tranzakcii/>.
- [51] Что такое Proof-of-Work и Proof-of-Stake? // <https://trendcoin.ru/cto-takoe-proof-of-work-i-proof-of-stake/>.
- [52] Чотакое Segregated Witness (SegWit)// <https://bitnovosti.com/2021/11/17/cto-takoe-segregated-witness-segwit/> – 17.11.2021.
- [53] Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке С / пер. с англ. М.: ТРИУМФ, 2002. 816 с.
- [54] Шустов Д. Алгоритм консенсуса Tendermint – полный обзор // <https://ex4.ru/technologies/consensus/tendermint> – 09.07.2019.
- [55] Andresen G. BIP34. Block v2, Height in Coinbase // https://en.bitcoin.it/wiki/BIP_0034 – 2012-07-06.
- [56] Andresen G. Blockchain Rule Update Process // <https://gist.github.com/gavinandresen/2355445>.
- [57] Andresen G., Hearn M. BIP70. Payment Protocol // <https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki> – 2013-07-29.
- [58] Baird L. Hashgraph Consensus: Detailed Examples // <https://www.swirlds.com/downloads/SWIRLDS-TR-2016-02.pdf> – Swirlds tech report – Revision date February 20, 2018.
- [59] Bakhtiari S., Safavi-Naini R., Pieprzyk J. Practical and Secure Message Authentication // <http://citeseerx.ist.psu.edu> – April 27, 1995 – University of Wollongong.
- [60] Barker E., Roginsky A. NIST Special Publication 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths // <http://csrc.nist.gov> – National Institute of Standards and Technology – January 2011.
- [61] Base58Check encoding // https://en.bitcoin.it/wiki/Base58Check_encoding.
- [62] Bellare M., Canetti R., Krawczyk H. Keying Hash Functions for Message Authentication // <http://cseweb.ucsd.edu> – June 1996.

- [63] *Bellare M., Kohno T.* Hash Function Balance and its Impact on Birthday Attacks // <http://eprint.iacr.org> – May 2004 – University of California at San Diego, La Jolla, California, USA.
- [64] *Benhamouda F., Lepoint T., Loss J., Orrù M., Raykova M.* On the (in)security of ROS // Cryptology ePrint Archive: Report 2020/945 – 2020.
- [65] *Beregszaszi A., Bylica P.* EIP 145: Bitwise shifting instructions in EVM // <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-145.md> – 2017-02-13.
- [66] *Bernstein D. J.* Understanding brute force // <http://cr.yp.to> – 2005.04.25 – The University of Illinois at Chicago.
- [67] *Bernstein D. J., Birkner P., Joye M., Lange T., Peters C.* Twisted Edwards Curves // Cryptology ePrint Archive: Report 2008/013 – 2008.
- [68] *Bernstein D. J., Duif N., Lange T., Schwabe P., Yang B.-Y.* High-Speed High-Security Signatures // Journal of Cryptographic Engineering – 2012 – No. 2 – pp. 77–89.
- [69] *Bertoni G., Daemen J., Peeters M., Van Assche G.* Cryptographic sponge functions // <http://sponge.noekeon.org> – Version 0.1 – January 14, 2011.
- [70] *Bertoni G., Daemen J., Peeters M., Van Assche G.* Keccak specifications // <http://keccak.noekeon.org> – October 27, 2008.
- [71] *Bertoni G., Daemen J., Peeters M., Van Assche G.* Keccak sponge function family main document // <http://keccak.noekeon.org> – Version 1.2 – April 23, 2009.
- [72] *Biham E., Chen R.* Near-Collisions of SHA-0 // <http://eprint.iacr.org> – Technion, Haifa, Israel.
- [73] *Biham E., Chen R., Joux A., Carribault P., Lemuet C., Jalby W.* Collisions of SHA-0 and Reduced SHA-1 // <http://www.iacr.org>.
- [74] *Biham E., Shamir A.* Differential Cryptanalysis of DES-like Cryptosystems // <http://citeseer.ist.psu.edu> – The Weizmann Institute of Science, Israel – July 19, 1990.
- [75] *Biham E., Shamir A.* Differential Cryptanalysis of the full 16-round DES // <http://citeseer.ist.psu.edu> – Technion, Haifa, Israel – 1991.
- [76] BIP39 Wordlists // <https://github.com/bitcoin/bips/blob/master/bip-0039/bip-0039-wordlists.md>.
- [77] *Bistarelli S., Mercanti I., Santini F.* An Analysis of Non-standard Transactions // <https://www.frontiersin.org/articles/10.3389/fbloc.2019.00007/full>.
- [78] Bitcoin Core version 0.9.0 released // <https://bitcoin.org/en/release/v0.9.0>.
- [79] Bitcoin Developer. Blockchain // https://developer.bitcoin.org/devguide/block_chain.html.
- [80] Bitcoin Developer. Wallets // <https://developer.bitcoin.org/devguide/wallets.html>.
- [81] BitCoin Forum [Full Disclosure] CVE-2012-2459 (block merkle calculation exploit) // <https://bitcointalk.org/?topic=102395>.
- [82] Bitcoin Trust Platform // <https://www.bitrated.com/>.
- [83] Bitcoinj // <https://bitcoinj.org>.
- [84] *Blaze M., Diffie W., Rivest R. L., Schneier B., Shimomura T., Thompson E., Wiener M.* Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security // <http://www.schneier.com> – January 1996.
- [85] *Boldyreva A.* Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme // Desmedt Y. G. (Ed.) Public-Key Cryptography – PKC 2003, LNCS 2567, pp. 31–46, 2003.

- [86] *Boneh D., Drijvers M., Neven G.* BLS Multi-Signatures With Public-Key Aggregation // <https://crypto.stanford.edu> – March 24, 2018.
- [87] *Boneh D., Lynn D., Shacham H.* Short Signatures from the Weil Pairing // *Boyd C. (Ed.): Advances in Cryptology – ASIACRYPT 2001, LNCS 2248*, pp. 514–532, 2001.
- [88] *Borst J., Preneel B., Vandewalle J.* On the Time-Memory Tradeoff Between Exhaustive Key Search and Table Precomputation // <http://citeseerx.ist.psu.edu> – K. U. Leuven, Heverlee, Belgium.
- [89] *Branstad D.* Proposed NIST SHS // <http://w2.eff.org>.
- [90] *Brown D. R. L.* Standards for Efficient Cryptography. SEC 2: Recommended Elliptic Curve Domain Parameters // <http://www.secg.org/sec2-v2.pdf>.
- [91] *Buterin V.* Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform // <https://ethereum.org/en/whitepaper> – 2014.
- [92] *Castro M., Liskov B.* Practical Byzantine Fault Tolerance // eProceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999. Available at <http://pmg.csail.mit.edu/papers/osdi99.pdf>.
- [93] *Chabaud F., Joux A.* Differential Collisions in SHA-0 // <http://books.google.ru> – 1998 – Centre d’Électronique de l’Armement, Rennes Armées, France.
- [94] Chainlist. Helping users connect to EVM powered networks // <https://chainlist.org>.
- [95] *Chang S., Perlner R., Burr W. E., Turan M. S., Kelsey J. M., Paul S., Bassham L. E.* Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition // <http://nvlpubs.nist.gov> – National Institute of Standards and Technology – November 2012.
- [96] *Cochran M.* Notes on the Wang et al. 2^{63} SHA-1 Differential Path // <http://eprint.iacr.org> – August 24, 2008 – Boulder, Colorado, USA.
- [97] *Coelho I. M., Coelho V. N., Araujo R. P., Wang Y. Q., Rhodes B. D.* Challenges of PBFT-Inspired Consensus for Blockchain and Enhancements over Neo dBFT // Future Internet 2020, vol. 12, issue 8, 129.
- [98] Common Vulnerabilities and Exposures // https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2012-2459.
- [99] Contracts – Bitcoin // <https://developer.bitcoin.org/devguide/contracts.html> – Bitcoin Developer.
- [100] *Courtois N. T.* Is AES a Secure Cipher? // <http://www.cryptosystem.net>.
- [101] *Daemen J., Peeters M., Van Assche G., Rijmen V.* Nessie Proposal: Noekeon // <http://www.cosic.esat.kuleuven.be> – 2000.
- [102] *Damgård I. B.* A Design Principle of Hash Functions // In G. Brassard (Ed.): Advances in Cryptology – CRYPTO’89, LNCS 435, pp. 416–427, 1990 – Aarhus University, Aarhus, Denmark.
- [103] *De Cannière C., Rechberger C.* Preimages for Reduced SHA-0 and SHA-1 // <http://online.tu-graz.ac.at>.
- [104] *Den Boer B., Bosselaers A.* An Attack on the Last Two Rounds of MD4 // <http://citeseerx.ist.psu.edu> – 15 October 1991.
- [105] *Diffie W., Hellman M. E.* New directions in cryptography // IEEE Transactions on Information Theory, vol. IT-22, no. 6, pp. 644–654, November 1976.
- [106] *Dobbertin H., Bosselaers A., Preneel B.* RIPEMD-160: A Strengthened Version of RIPEMD // <http://homes.esat.kuleuven.be> – 18 April 1996.

- [107] *Driscoll K., Hall B., Sivencrona H., Zumsteg P.* Byzantine Fault Tolerance, from Theory to Reality // Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23–26, 2003, Proceedings. Available at https://www.researchgate.net/publication/226020071_Byzantine_Fault_Tolerance_from_Theory_to_Reality.
- [108] *Dusse S. R., Kaliski B. S. Jr.* A Cryptographic Library for the Motorola DSP56000 // <http://ftp.zedz.net> – 1998 – RSA Data Security Inc.
- [109] *Dwork C., Naor M.* Pricing via Processing or Combatting Junk Mail // Advances in Cryptology – CRYPTO'92, Proceedings, pp. 139–147.
- [110] eBACS: ECRYPT Benchmarking of Cryptographic Systems. Introduction // <http://bench.cr.yp.to> – 2010.09.03.
- [111] eBASH: ECRYPT Benchmarking of All Submitted Hashes // <http://bench.cr.yp.to> – 2008.11.02.
- [112] *El Gamal T.* A public key cryptosystem and a signature scheme based on discrete logarithms // Advances in Cryptology – Proceedings of CRYPTO 84, Springer-Verlag, 1985.
- [113] *El Mrabet N., Joye M. (Eds.)* Guide to Pairing-Based Cryptography. Chapman & Hall/CRC Press, 2016.
- [114] EOS vs Ethereum: консенсус, смарт-контракты, управление // <https://forklog.com/eos-vs-ethereum-konsensus-smart-kontrakty-upravlenie/>.
- [115] EOSIO Developer Portal // <https://developers.eos.io>.
- [116] ERC-20 Token Standard // <https://ethereum.org/ru/developers/docs/standards/tokens/erc-20/>.
- [117] Ethereum Casper: A Comprehensive Guide // <https://www.skalex.io/ethereum-casper/>.
- [118] Ethereum Constantinople & St. Petersburg Hard Forks // <https://medium.com/okex-blog/ethereum-constantinople-st-petersburg-hard-forks-26cf51a522ab>.
- [119] Ethereum Improvement Proposals // <https://eips.ethereum.org>.
- [120] Everything you need to know about the Ethereum «hard fork» // <https://qz.com/730004/everything-you-need-to-know-about-the-ethereum-hard-fork/>.
- [121] Explicit-Formulas Database // <http://www.hyperelliptic.org/EFD/index.html>.
- [122] *Ferguson N., Schroeppel R., Whiting D.* A simple algebraic representation of Rijndael // <http://citeseer.ist.psu.edu> – Draft 2001/05/16.
- [123] *Fiat A., Naor M.* Rigorous Time/Space Tradeoffs for Inverting Functions // <http://citeseerx.ist.psu.edu> – 1991.
- [124] FIPS 46-3. Data Encryption Standard (DES) // <http://csrc.nist.gov> – Reaffirmed 1999 October 25.
- [125] FIPS PUB 180. Secure Hash Standard // <http://csrc.nist.gov> – National Institute of Standards and Technology – 1993 May 11.
- [126] FIPS PUB 180-1. Secure Hash Standard // <http://www.itl.nist.gov> – National Institute of Standards and Technology – 1995 April 17.
- [127] FIPS PUB 180-2. Secure Hash Standard // <http://www.securitytechnet.com> – National Institute of Standards and Technology – 2002 August 1.
- [128] FIPS PUB 180-2. Secure Hash Standard. Change Notice 1 // <http://www.securitytechnet.com> – National Institute of Standards and Technology, Gaithersburg, MD – 2004 February 25.

- [129] FIPS PUB 180-4. Secure Hash Standard // <http://csrc.nist.gov> – National Institute of Standards and Technology – March 2012.
- [130] FIPS PUB 186. Digital Signature Standard (DSS) // <http://www.itl.nist.gov> – National Institute of Standards and Technology – 1994 May 19.
- [131] FIPS PUB 197. Specification for the Advanced Encryption Standard // <http://csrc.nist.gov> – National Institute of Standards and Technology – November 26, 2001.
- [132] FIPS PUB 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions // <http://csrc.nist.gov> – National Institute of Standards and Technology, Gaithersburg, MD – August 2015.
- [133] Forth (programming language) // https://en.wikipedia.org/wiki/Forth_%28programming_language%29.
- [134] Free Rainbow Tables. Distributed Rainbow Table Project // <http://www.freerainbowtables.com>.
- [135] *Fuller J., Millan W.* On Linear Redundancy in the AES S-box // <http://eprint.iacr.org> – 2002 – Queensland University of Technology, Brisbane, Australia.
- [136] Ganache – One click blockchain // <https://trufflesuite.com/ganache>.
- [137] Gas and fees // <https://ethereum.org/en/developers/docs/gas/>.
- [138] *Gaul A., Khoffi I., Liesen J., Stüber T.* Mathematical Analysis and Algorithms for Federated Byzantine Agreement Systems // https://www.researchgate.net/publication/337730065_Mathematical_Analysis_and_Algorithms_for_Federated_Byzantine_Agreement_Systems.
- [139] *Gauravaram P.* Cryptographic Hash Functions: Cryptanalysis, Design and Applications // <http://eprints.qut.edu.au> – Queensland University of Technology – 2007.
- [140] *Gennaro R., Jarecki S., Krawczyk H., Rabin N.* Secure Distributed Key Generation for Discrete-Log Based Cryptosystems // Journal of Cryptology – 2007 – No. 20 – pp. 51–83.
- [141] Get Started with Docker // <https://www.docker.com/get-started>.
- [142] *Girault M., Cohen R., Campana M.* A Generalized Birthday Attack // <http://dsns.csie.nctu.edu.tw>.
- [143] Go Ethereum // <https://geth.ethereum.org>.
- [144] *Hellman M. E.* A Cryptanalytic Time – Memory Trade-Off // <http://www-ee.stanford.edu> – October 24, 1978 – Stanford University, CA.
- [145] *Hill J. E.* Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard // <http://csrc.nist.gov> – National Institute of Standards and Technology – Federal Register, Vol. 72, No. 14, pp. 2861–2863 – January 23, 2007.
- [146] *Hong J., Jeong K. C., Kwon E. Y., Lee I.-S., Ma D.* Variants of the Distinguished Point Method for Cryptanalytic Time Memory Trade-offs (Full version) // <http://eprint.iacr.org> – 2008 – Seoul National University, Korea.
- [147] How Rainbow Tables work // <http://kestas.kuliukas.com>.
- [148] Hyperledger Architecture, Volume 1 // https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.
- [149] Hyperledger Foundation. HL_Greenhouse_Current // https://www.hyperledger.org/use/attachment/hl_greenhouse_current.

- [150] IPChain Смарт-контракт // <http://rspp.ru/upload/iblock/2f9/IPChainСмарт-контракты.pdf>.
- [151] Johnson N., Bylica P. EIP-1052: EXTCODEHASH opcode // <https://eips.ethereum.org/EIPS/eip-1052> – 2018-05-02.
- [152] Josefsson S., Liusvaara I. RFC 8032. Edwards-Curve Digital Signature Algorithm (EdDSA) // <http://tools.ietf.org> – January 2017.
- [153] Joux A. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions // <http://wb.cecs.pdx.edu> – 2004 – DCSSI Crypto Lab, Paris, France.
- [154] Jueneman R. R. A High Speed Manipulation Detection Code // <http://dsns.csie.nctu.edu.tw> – Computer Sciences Corp., Falls Church, VA.
- [155] Kaliski B. RFC 1319. The MD2 Message-Digest Algorithm // <http://tools.ietf.org> – April 1992 – RSA Laboratories.
- [156] Kayser R. F. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family // <http://csrc.nist.gov> – National Institute of Standards and Technology – Federal Register, Vol. 72, No. 212, pp. 62 212–62 220 – November 2, 2007.
- [157] Kelsey J., Schneier B., Wagner D. Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES // <http://www.schneier.com> – 1996.
- [158] Kim J., Biryukov A., Preneel B., Hong S. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0, and SHA-1 // <http://eprint.iacr.org> – 2006.
- [159] Knudsen L. R. A Key-schedule Weakness in SAFER K-64 // <http://citeseer.ist.psu.edu> – 1995 – École Normale Supérieure, Paris, France.
- [160] Knudsen L. R., Mathiassen J. E. Preimage and Collision Attacks on MD2 // <http://www.ii.uib.no>.
- [161] Knudsen L. R., Mathiassen J. E., Muller F., Thomsen S. S. Cryptanalysis of MD2 // Journal of Cryptology, 2010, 23, pp. 72–90.
- [162] Komlo C., Goldberg I. FROST: Flexible Round-Optimized Schnorr Threshold Signatures // Cryptology ePrint Archive: Report 2020/852 – 2020.
- [163] Krawczyk H., Bellare M., Canetti R. RFC 2104. HMAC: Keyed-Hashing for Message Authentication // <http://tools.ietf.org> – February 1997.
- [164] Langley A., Hamburg M., Turner S. RFC 7748. Elliptic Curves for Security // <http://tools.ietf.org> – January 2016.
- [165] Lenstra A. K., Lenstra H. W. (Eds.) The Development of the Number Field Sieve, Springer-Verlag, 1993.
- [166] Leurent G. MD4 is Not One-Way // <http://www.di.ens.fr> – École Normale Supérieure, Paris, France.
- [167] Linn J. RFC 1115. Privacy Enhancement for Internet Electronic Mail: Part III – Algorithms, Modes, and Identifiers // <http://tools.ietf.org> – August 1989 – DEC.
- [168] Lombrozo E., Lau J., Wuille P. BIP141. Segregated Witness (Consensus layer) // <https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki> – 2015-12-21.
- [169] Manuel S., Peyrin T. Collisions on SHA-0 in one hour // <http://fse2008.epfl.ch>.
- [170] Mao M., Chen S., Xu J. Construction of the Initial Structure for Preimage Attack of MD5 // <http://www.lw20.com> – 11–14 Dec. 2009 – Univ. of Electron. Sci. & Technol. of China, Chengdu, China.

- [171] *McDonald C., Hawkes P., Pieprzyk J.* Differential Path for SHA-1 with complexity $O(2^{52})$ // <http://eprint.iacr.org>.
- [172] *Mendel F., Pramstaller N., Rechberger C., Kontak M., Szmidt J.* Cryptanalysis of the GOST Hash Function // <http://eprint.iacr.org>.
- [173] *Menezes A., Okamoto T., Vanstone S.* Reducing Elliptic Curve Logarithms in a Finite Field // IEEE Transactions on Information Theory – vol. IT-39 (1993) – No. 5 – pp. 1639–1646.
- [174] *Menezes A., van Oorschot P., Vanstone S.* Handbook of Applied Cryptography. CRC Press, 1996.
- [175] *Menezes A., Sarkar P., Singh S.* Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography // Cryptology ePrint Archive: Report 2016/1102 – 2016.
- [176] *Mentens N., Batina L., Preneel B., Verbauwheide I.* Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking // <http://www.cosic.esat.kuleuven.be> – 2006 – Katholieke Universiteit Leuven, Leuven-Heverlee, Belgium.
- [177] *Merkle R. C.* A Certified Digital Signature // *Brassard G. (Ed.)* Advances in Cryptology – CRYPTO'89, LNCS 435, pp. 218–238, 1990 – Xerox PARC, Palo Alto, Ca.
- [178] *Merkle R. C.* U. S. Patent # 4 309 569. Method of providing digital signatures. 1982-01-05.
- [179] MetaMask – A crypto wallet & gateway to blockchain apps // <https://metamask.io>.
- [180] Microsoft Docs. Установка WSL // <https://docs.microsoft.com/ru-ru/windows/wsl/install> – 08.04.2022.
- [181] Mini private key format // https://en.bitcoin.it/wiki/Mini_private_key_format.
- [182] *Miyaguchi S., Ohta K., Iwata M.* 128-bit hash function (N-hash) // NTT Review – 1990 – No. 2 – pp. 128–132.
- [183] Mnemonic Code Converter // <https://iancoleman.io/bip39/>.
- [184] *Montgomery P. L.* Speeding the Pollard and Elliptic Curve Methods of Factorization // Mathematics of Computation – vol. 48 (1987) – No. 177 – pp. 243–264.
- [185] MultiChain for Developers // <https://www.multichain.com/developers/>.
- [186] *Murphy S., Robshaw M. J. B.* Essential Algebraic Structure Within the AES // <http://citeseer.ist.psu.edu> – University of London, Egham, U. K.
- [187] *Nakamoto S.* Bitcoin: A Peer-to-Peer Electronic Cash System // <https://bitcoin.org/bitcoin.pdf>.
- [188] *Nandi M., Stinson D. R.* Multicollision Attacks on Generalized Hash Functions // <http://citeseerx.ist.psu.edu>.
- [189] Neo Documentation // <https://docs.neo.org/docs/en-us/index.html>.
- [190] NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition // <http://www.nist.gov> – Information Technology Laboratory – October 2, 2012.
- [191] *Oechslin P.* Making a Faster Cryptanalytic Time-Memory Trade-Off // <http://lasecwww.epfl.ch> – École Polytechnique Fédérale de Lausanne, Switzerland.
- [192] *Oechslin P.* OPHCRACK (the time-memory-trade-off-cracker) // <http://lasecwww.epfl.ch> – April 3rd 2006 – École Polytechnique Fédérale de Lausanne, Switzerland.

- [193] *Palasz C.* Understanding BIP39 and Your Mnemonic Phrase // <https://privacypros.io/wallets/mnemonic-phrase>.
- [194] *Palatinus M., Rusnak P., Voisine A., Bowe S.* BIP39. Mnemonic code for generating deterministic keys // <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki> – 2013-09-18.
- [195] *Pedersen T. P.* A Threshold Cryptosystem without a Trusted Party // *Davies D. W.* (Ed.) Advances in Cryptology – EUROCRYPT’91, LNCS 547, pp. 522–526, 1991.
- [196] *Peyrin T.* Collisions in SHA-0 in one hour // <http://thomas.peyrin.googlepages.com> – December 13, 2007 – IPA Cryptographic Workshop 2007 – Tokyo, Japan.
- [197] *Pohlig S. C., Hellman M. E.* An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance // IEEE Transactions on Information Theory – vol. 1 (1978) – No. 24 – pp. 106–110.
- [198] *Pollard J. M.* Monte Carlo Methods for Index Computation (mod p) // Mathematics of Computation – vol. 32 (1978) – No. 143 – pp. 918–924.
- [199] Polys от «Лаборатории Касперского» представляет блокчейн-машины для голосования // https://www.kaspersky.ru/about/press-releases/2020_polys-ot-laboratorii-kasperskogo-predstavlyaet-blokchein-mashini-dlya-golosovaniya – 27 февраля 2020 г. – АО «Лаборатория Касперского».
- [200] *Popov S.* The Tangle // <https://www.semanticscholar.org/paper/The-tangle-Popov/feb18bc2793e907e9404ffe36e442162cd2b2688>.
- [201] *Popov V., Kurepin I., Leontiev S.* RFC 4357. Additional Cryptographic Algorithms for Use with GOST 28147–89, GOST R 34.10–94, GOST R 34.10–2001, and GOST R 34.11–94 Algorithms // <http://tools.ietf.org> – January 2006 – CRYPTO-PRO.
- [202] *Preneel B.* Analysis and Design of Cryptographic Hash Functions // <http://homes.esat.kuleuven.be> – February 2003.
- [203] *Preneel B., van Oorschot P. C.* MDx-MAC and Building Fast MACs from Hash Functions // <ftp://ftp.esat.kuleuven.be> – August 1995.
- [204] Python Bitcoin Library // <https://coineva.com/python-bitcoin-library.html>.
- [205] *Quisquater J.-J., Delescaille J.-P.* How easy is collision search? Application to DES (Extended summary) // <http://dsns.csie.nctu.edu.tw> – Philips Research Laboratory, Louvain-la-Neuve, Belgium.
- [206] RainbowCrack Tutorial // <http://project-rainbowcrack.com>.
- [207] Remix IDE // <https://remix.ethereum.org>.
- [208] Rinkeby: Network Dashboard // <https://www.rinkeby.io>.
- [209] Ripple // <https://ripple.com/xrp>.
- [210] *Rivest R.* RFC 1186. The MD4 Message Digest Algorithm // <http://www.ietf.org> – MIT Laboratory for Computer Science – October 1990.
- [211] *Rivest R.* RFC 1320: The MD4 Message-Digest Algorithm // <http://www.ietf.org> – April 1992.
- [212] *Rivest R.* RFC 1321. The MD5 Message Digest Algorithm // <http://www.ietf.org> – April 1992.
- [213] *Rivest R.* The MD6 Hash Function (aka «Pumpkin Hash») // <http://groups.csail.mit.edu> – MIT – CRYPTO 2008.
- [214] *Rivest R. L., Agre B., Bailey D. V., Crutchfield C., Dodis Y., Fleming K. E., Khan A., Krishnamurthy J., Lin Y., Reyzin L., Shen E., Sukha J., Sutherland D., Tromer E.*

- Yin Y. L.* The MD6 hash function. A proposal to NIST for SHA-3 // <http://csrc.nist.gov> – October 27, 2008 – Massachusetts Institute of Technology, Cambridge, MA.
- [215] Ropsten Faucet // <https://faucet.ropsten.be>.
- [216] Ropsten testnet PoW chain // <https://github.com/ethereum/ropsten>.
- [217] *Sasaki Y.* Collisions of MMO-MD5 and Their Impact on Original MD5 // <http://link.springer.com> – 2011.
- [218] *Sasaki Y., Aoki K.* Finding Preimages in Full MD5 Faster than Exhaustive Search // <http://www.iacr.org> – NTT Corporation, Tokyo, Japan.
- [219] *Sasaki Y., Wang L., Ohta K., Kunihiro N.* New Message Difference for MD4 // <http://www.iacr.org> – The University of Electro-Communications, Tokyo, Japan.
- [220] Schneier on Security: NIST Hash Workshop Liveblogging // <http://www.schneier.com> – November 1, 2005.
- [221] *Schoedon A.* EIP-1234: Constantinople Difficulty Bomb Delay and Block Reward Adjustment // <https://eips.ethereum.org/EIPS/eip-1234> – 2018-07-19.
- [222] Script // <https://en.bitcoin.it/wiki/Script>.
- [223] SHA hash functions // <http://www.ursalab.com> – UrsaLab Software.
- [224] *Shamir A.* How to Share a Secret // Communications of the ACM – vol. 22 (1979) – No. 11.
- [225] *Smart N. P.* The Discrete Logarithm Problem on Elliptic Curves of Trace One // Journal of Cryptology – 1999 – No. 12 – pp. 193–196.
- [226] *Smyshlyayev S. (Ed.), Alekseev E., Oshkin I., Popov V., Leontiev S., Podobaev V., Belyavsky D.* RFC 7836. Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10–2012 and GOST R 34.11–2012 // <http://tools.ietf.org> – March 2016.
- [227] Solidity // <https://docs.soliditylang.org/en/latest/>.
- [228] *Standaert F.-X., Rouvroy G., Quisquater J.-J., Legat J.-D.* FPGA Implementation of a Cryptanalytic Time-Memory tradeoff // <http://www.elis.rug.ac.be> – Université Catholique de Louvain, Louvain-La-Neuve, Belgium.
- [229] Stellar // <https://www.stellar.org/>.
- [230] *Stevens M., Sotirov A., Appelbaum J., Lenstra A., Molnar D., Osvik D. A., de Weger B.* Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate // <http://marc-stevens.nl> – 2009.
- [231] *Sugita M., Kawazoe M., Imai H.* Gröbner Basis Based Cryptanalysis of SHA-1 // <http://citeseer.ist.psu.edu>.
- [232] *Szabo N.* The Idea of Smart Contracts // <https://nakamotoinstitute.org/the-idea-of-smart-contracts/>.
- [233] *Tang W.* Net gas metering for SSTORE without dirty maps // <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1283.md> – 2018-08-01.
- [234] The Corda Platform: An Introduction White Paper // <https://www.r3.com/white-papers/the-corda-platform-an-introduction-whitepaper/>.
- [235] The Ethereum Blockchain Explorer // <https://etherscan.io>.
- [236] The MD6 Hash Algorithm // <http://groups.csail.mit.edu>.
- [237] *Turner S., Chen L.* RFC 6150. MD4 to Historic Status // <http://tools.ietf.org> – March 2011.
- [238] Uniswap Protocol // <https://uniswap.org>.

- [239] Use Nulldata to store data on the Bitcoin Blockchain // <https://coineva.com/use-nulldata-to-send-blockchain-messages.html>.
- [240] *Van Oorschot P. C., Wiener M. J.* Parallel Collision Search with Application to Hash Functions and Discrete Logarithms // <http://www.scs.carleton.ca> – 1994 August 17 – Bell-Northern Research, Ottawa, Ontario, Canada.
- [241] *Van Rompay B., Preneel B., Vandewalle J.* On the security of dedicated hash functions // <http://citeseerx.ist.psu.edu> – Katholieke Universiteit Leuven, Heverlee, Belgium.
- [242] *Wang X.* Cryptanalysis on Hash Functions // <http://www.cvicse.com> – 10/28/2005.
- [243] *Wang X., Feng D., Lai X., Yu H.* Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD // <http://eprint.iacr.org> – revised on August 17, 2004.
- [244] *Wang X., Yu H.* How to Break MD5 and Other Hash Functions // <http://citeseerx.ist.psu.edu> – Shandong University, Jinan, China.
- [245] *Wheeler D. J., Needham R. M.* TEA, a Tiny Encryption Algorithm // <http://www.cl.cam.ac.uk> – Computer Laboratory, Cambridge University, England.
- [246] Wikipedia, the free encyclopedia // <http://en.wikipedia.org>.
- [247] Working with micropayment channels // <https://bitcoinj.org/working-with-micropayments>.
- [248] *Wuille P.* BIP30. Duplicate Transactions // <https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki> – 2012-02-22.
- [249] *Wuille P.* BIP62. Dealing with malleability // <https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki> – 2014-03-12.
- [250] *Yoshida H., Biryukov A., Preneel B.* Some applications of the Biham-Chen attack to SHA-like hash functions // <http://csrc.nist.gov> – Cryptographic Hash Workshop NIST, Gaithersburg, Maryland – October 31, 2005.
- [251] 1 мBTC – это сколько BTC? Чему равен 1 сатоши? Что такое сатоши? // <https://calcsbox.com/post/1-mbtc-eto-skolko-btc-cemu-raven-1-satosi-cto-takoe-satosi.html>.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «КТК Галактика» наложенным платежом,
выслав открытку или письмо по почтовому адресу:

115487, г. Москва, пр. Андропова д. 38 оф. 10.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.galaktika-dmk.com.

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: books@aliants-kniga.ru.

**Евгения Александровна Ищукова,
Сергей Петрович Панасенко,
Кирилл Сергеевич Романенко,
Вячеслав Дмитриевич Салманов**

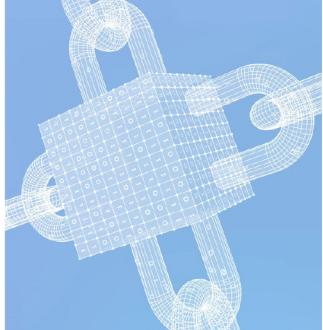
Криптографические основы блокчейн-технологий

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 24,54. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com



Построение блокчейн-систем

Любыми технологиями можно просто пользоваться, не вникая в их суть, а можно и стремиться к глубокому знанию внутренних механизмов и пониманию внутренних процессов, составляющих основу используемых решений.

Данная книга поощряет подобное стремление в отношении современных блокчейн-технологий и рассматривает вопросы их построения с основным упором на криптографические составляющие блокчейн-систем, включая широко используемые криптовалюты: Биткойн, Эфириум и другие.

Книга насыщена примерами и иллюстрациями, позволяющими лучше разобраться в материале.

Прочитав эту книгу, вы узнаете:

- какие криптографические алгоритмы применяются в современных блокчейн-платформах, их историю, структуру и особенности применения;
- как производится формирование транзакций, их объединение в блоки и связь блоков между собой в единую цепочку;
- основные механизмы распределения ответственности за формирование корректной цепочки между узлами системы и достижения консенсуса между ними;
- корректные методы построения и валидации транзакций в криптовалютных блокчейн-системах на примере системы Биткойн;
- возможности по разработке смарт-контрактов и их примеры для систем Биткойн, Эфириум и Hyperledger.

Издание может быть полезно как специалистам в области блокчейн-технологий, так и тем, кто только начинает интересоваться этой темой.

ISBN 978-5-97060-865-4



9 785970 608654 >

Интернет-магазин:

www.dmkpress.com

Оптовая продажа:

КТК "Галактика"

books@aliens-kniga.ru

