O'REILLY®

Python для программирования криптовалют

Как научиться программировать биткойн "с чистого листа"

Джимми Сонг

Python
для программирования криптовалют

Programming bitcoin

Jimmy Song

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Python
для программирования криптовалют

Джимми Сонг

Москва · Санкт-Петербург
2020

ББК 32.973.26-018.2.75
С62
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*
При участии *В.А. Коваленко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info.dialektika@gmail.com, http://www.dialektika.com

Сонг, Джимми.

С62 Python для программирования криптовалют. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 368 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-82-8 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джимми Сонг

Python для программирования криптовалют

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-82-8 (рус.)

ISBN 978-1-492-03149-9 (англ.)

Оглавление

Содержание

Об авторе

Джимми Сонг является разработчиком с более чем 20-летним стажем, начавший свою карьеру с чтения второго издания книги *Programming Perl*, вышедшей в том же самом издательстве, в котором вышла и данная книга. Он принимал участие во многих новых предприятиях и полностью занялся биткойном в 2014 году. С тех пор он внес свой посильный вклад в самые разные проекты биткойна с открытым кодом, включая Armory, Bitcoin Core, `btcd` и `pycoin`.

Если вы когда-нибудь повстречаете Джимми и захотите вызвать в нем желание порассуждать, спросите его о биткойне, устойчивой валюте, худых последствиях от декретной валюты, посте, плотоядии, силовой атлетике, воспитании детей и ковбойских шляпах.

Об изображении на обложке

На обложке настоящего издания этой книги изображен медоед (Mellivora capensis), иначе называемый лысым барсуком. Несмотря на свое название, это животное внешне похоже скорее на ласку или хорька, чем на барсука. Оно обитает повсюду в Африке, Индии и Юго-Западной Азии. Медоед — хищное животное, проявляющее свой невероятно свирепый нрав, когда ему приходится защищаться.

Свое привычное название медоед получил из-за повадки совершать набеги на ульи и поедать любимый им мед и личинки пчел. У него толстая шкура, сводящая на нет последствия пчелиных укусов. Но рацион медоедов весьма разнообразен и включает в себя змей (в том числе ядовитых), грызунов, насекомых, лягушек, птиц, яйца, плоды, корни и луковицы растений. Было замечено, что медоед прогоняет молодых львов от своей добычи и относится к немногим видам животных, наблюдения за которым велись с использованием современных инструментов.

Медоед — крепко сложенное животное с длинным телом, широкой спиной и небольшой плоской головой. У него короткие ноги, оканчивающиеся острыми когтями, благодаря которым он исключительно хорошо роет землю. Медоед способен не только докопаться до своей добычи под землей, но и вырыть нору для себя (длиной в среднем 1–3 м). У основания его хвоста находится железа, наполненная зловонной секрецией, которой он пользуется для пометки своей территории и предупреждения других животных. Шкура

на шее медоеда не натянута, что позволяет ему вертеть головой и кусаться, когда его хватают.

В 2011 году бесстрашное поведение медоеда стало сюжетом комического вирусного видео, снятого *National Geographic*.

Многие виды животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, хотя все они важны для нашего мира. Подробнее о том, как помочь спасению этих животных, можно узнать по адресу `animals.oreilly.com`. На первой странице обложки этой книги приведена иллюстрация с изображением медоеда, выполненная Карен Монтгомери на основании черно-белой гравюры, взятой из энциклопедического издания *Natural History of Animals* (Естественная история животных).

Предисловие

Как замечательно быть писателем научно-фантастических повестей и романов! В них можно построить общество, в котором благосостояние больше не является миражом, возникающим из пустых обещаний правительств и манипуляций центральных банков, обмениваться ценностями можно безо всяких подозрений и оплаты услуг посредника, код может быть законом, а коллективные решения могут приниматься без искажений, вносимых централизацией... Для этого писателю достаточно открыть текстовый редактор и приступить к воплощению замысла своего научно-фантастического детища.

Но для написания захватывающих романов нужно не только воображение, но и знание окружающего мира. "Построение мира" — это не столько литературное правдоподобие или напичканный техническими терминами текст, сколько проникновение глубоко внутрь, чтобы добраться до сути и понять, каким образом действует этот мир, задаваясь вопросами вроде "Что если..." И чем лучше писатель понимает механизмы и коды, образующие описываемый им мир, тем более интересными становятся вопросы, которые в связи с ними возникают.

Изменить реальный мир намного труднее, чем написать научно-фантастический роман, но и для этого требуются знания. Помимо мудрости, идеализма, настойчивости, дисциплины и убежденности перед лицом сомнений, тому, кто претендует на преобразование окружающего мира, необходимо ясно понимать, какие средства и возможности ему доступны и каковы их ограничения.

В настоящее время мир биткойна и блокчейна все еще остается по большей части фантастическим. Доморощенные знатоки, вселяющие надежды, не скупящиеся на похвалы и совсем не понимающие реального положения вещей, слышны намного громче и оказывают большее влияние, чем те, кто выполняет всю тяжелую работу, чтобы изменить окружающий мир. Привлекая на помощь техническую терминологию и актуальные тематические метки в социальных сетях, политически мотивированные

разглагольствования, основанные на страхе и схемах быстрого обогащения, призывают жаждать знаний.

Но, читая техническую документацию и обзорные статьи, понять блокчейн можно не больше, чем научиться открывать свое дело, посещая школу предпринимательства или наблюдая презентации, составленные в PowerPoint.

Для этого придется программировать.

Чтобы разобраться в какой-нибудь технологии, лучше всего создать с ее помощью нечто полезное. До тех пор, пока вы самостоятельно не запрограммируете основополагающие конструктивные блоки приложения на основе блокчейна, вы не уловите разницы между пустой рекламной шумихой и осуществимыми возможностями.

Эта книга служит наиболее эффективным и исчерпывающим источником информации для изучения биткойна и блокчейна через программирование. Опираясь на свой опыт и знания, ее автор, Джимми Сонг, проложил такой путь к изучению, который проведет читателя от математических основ биткойна до новейших расширений и разветвлений. И на этом пути читателю предоставляется возможность поупражняться на примерах, которые были проверены на реальных учащихся и не только научат принципам действия биткойна, но и позволят интуитивно понять изящество и прелесть данной технологии.

Но путь этот нелегкий. Даже имея такого наставника и приверженца биткойна, как Джимми Сонг, читателю не стоит уповать на то, что, пролистав эту книгу скуки ради в промежутках между просмотром развлекательных сериалов от компании Netflix, он сможет легко разобраться в данной технологии. Для этого придется немало потрудиться. Кратчайшего пути к ее освоению не существует, как, впрочем, и возможности кратко изложить ее суть. Но именно это обстоятельство как нельзя лучше отвечает основополагающему принципу биткойна: иметь личную заинтересованность и демонстрировать доказательство выполнения работы. И лишь тогда можно доверять своим знаниям.

Удачного программирования!

— Кен Лю[1]

[1] **Кен Лю** (Ken Liu) — автор эпической фантастической серии книг *The Dandelion Dynasty* (*Династия одуванчиков*) в стиле "силкпанк" (шелковая чепуха), в которой техника является волшебством, а также сборника повестей *The Paper Menagerie and Other Stories* (*Бумажный зверинец и другие истории*). Его научно-фантастический рассказ *Byzantine Empathy* (*Византийская эмпатия*) о блокчейне был первоначально опубликован в издательстве MIT Press.

Введение

Эта книга призвана обучить технологии биткойна на базовом уровне. В ней не рассматривается монетарная, экономическая или социальная динамика развития биткойна, но поясняется внутренний механизм его действия, позволяющий лучше понять возможности данной технологии. В настоящее время наблюдается тенденция широко расхваливать биткойн и блокчейн, не понимая сути дела. И цель этой книги — противостоять подобной тенденции.

Существует немало литературы, в которой излагается история развития, техническое описание биткойна и экономические особенности его применения. А эта книга призвана помочь вам разобраться в биткойне, программируя все компоненты для его библиотеки. Назначение такой библиотеки — помочь вам в изучении биткойна, а не быть исчерпывающей или эффективной.

Для кого написана эта книга

Эта книга адресована тем программистам, которые стремятся изучить принцип действия биткойна, программируя его самостоятельно. Читателям представится возможность изучить биткойн, программируя "голое железо" с помощью библиотеки биткойна, которую нужно будет создать заново. Поэтому данную книгу не стоит рассматривать как справочное пособие, к которому можно обратиться за описанием конкретного функционального средства.

Материал этой книги основывается главным образом на материале двухдневного семинара (`https://programmingbitcoin.com/schedule/`), на котором разработчики могут научиться всему, что следует знать о биткойне. Этот материал существенно усовершенствовался по мере того, как данный курс обучения был проведен 20 раз более чем для 400 обучающихся на момент написания данной книги.

Завершив чтение этой книги, вы будете уметь не только создавать транзакции, но и получать все необходимые данные от партнеров и отправлять транзакции по сети. Здесь есть все, что для этого требуется, включая

математический аппарат, синтаксический анализ, связность узлов сети и проверку достоверности блоков.

Что нужно знать

Предпосылкой для чтения этой книги служит умение программировать, в частности, на языке Python. Сама рассматриваемая здесь библиотека написана на языке Python версии 3, а многие предлагаемые упражнения могут быть выполнены в такой интерактивной среде, как, например, командная оболочка Jupyter Notebook. Знать язык Python желательно на среднем уровне, хотя и начальных знаний может быть достаточно для усвоения многих понятий.

Для усвоения материала книги, в особенности глав 1 и 2, необходимо немного знать математику. В этих главах представлены математические понятия, которые, вероятно, малознакомы тем, кто не является знатоком математики. Впрочем, знаний математики на уровне алгебры должно быть достаточно, чтобы усвоить новые понятия и выполнить упражнения по программированию, предлагаемые в этих главах.

При чтении этой книги пригодится знание общих основ вычислительной техники (например, функций хеширования), хотя для выполнения предлагаемых здесь упражнений наличие таких знаний не строго обязательно.

Как организована эта книга

Эта книга разделена на 14 глав. Материал каждой последующей главы построен на материале предыдущей и позволяет создать библиотеку для биткойна от начала и до конца.

Грубо говоря, в главах 1–4 описывается весь требующийся математический аппарат. В главах 5–8 рассматриваются транзакции, являющиеся основными единицами биткойна, а в главах 9–12 — блоки и организация сети. И наконец, в главах 13 и 14 обсуждается ряд дополнительных вопросов, фактически не требующих написания кода.

А точнее говоря, необходимый математический аппарат излагается в главах 1, "Конечные поля", и 2, "Эллиптические кривые". В частности, для понимания основ криптографии на эллиптических кривых требуется знать, что такое конечные поля и эллиптические кривые, поясняемые в главе 3, "Криптография по эллиптическим кривым". И как только в конце главы 3 будет введено понятие криптографии с открытым ключом, в главе 4, "Сериализация",

будут представлены механизмы синтаксического анализа и сериализации, с помощью которых сохраняются и передаются криптографические примитивы.

В главе 5, "Транзакции", описывается структура транзакции, а в главе 6, "Язык Script" — язык используемый для написания умных контрактов, положенный в основу биткойна. В главе 7, "Создание и проверка достоверности транзакций", основывающейся на материале предыдущих глав, поясняется, как проверять на достоверность и создавать транзакции, применяя криптографию на эллиптических кривых, описанную в главах 1–4. А в главе 8, "Оплата по хешу сценария", разъясняется принцип действия оплаты по хешу сценария (p2sh) как способа повышения эффективности умных контрактов.

В главе 9, "Блоки", описываются блоки, представляющие собой группы упорядоченных транзакций, а в главе 10, "Организация сети" — передача данных по сети в биткойне. В главах 11, "Упрощенная проверка оплаты", и 12, "Фильтры Блума", поясняется, как "тонкий" клиент (т.е. прикладная программа без доступа ко всему блокчейну) может запросить и организовать взаимообмен данными с теми узлами, в которых хранится весь блокчейн.

В главе 13, "Протокол Segwit", описывается обратно совместимое обновление протокола, называемое Segwit и внедренное в 2017 году, а в главе 14, "Дополнительные вопросы и следующие шаги", даются рекомендации по дальнейшему изучению биткойна. И хотя читать эти главы совсем необязательно, они включены в книгу для того, чтобы дать некоторое представление о дальнейших путях изучения данной технологии.

Подготовка

Чтобы извлечь наибольшую пользу из этой книги, вам придется создать среду, в которой вы сможете выполнять примеры кода и упражнения. Ниже перечислены действия, которые следует предпринять, чтобы хорошо подготовиться к проработке материала этой книги.

1. Установите на своем компьютере интерпретатор языка Python не ниже версии 3.5, загрузив его дистрибутив по указанным ниже адресам.
   Для системы Windows — по адресу `https://www.python.org/ftp/python/3.6.2/python-3.6.2-amd64.exe`.
   Для macOS — по адресу `https://www.python.org/ftp/python/3.6.2/python-3.6.2-macosx10.6.pkg`.

Для систем Linux — обратитесь к документации на отдельные дистрибутивы (многие дистрибутивы Linux вроде Ubuntu поставляются вместе с предварительно установленной версией Python 3.5+).

2. Установите сценарий `pip`, загрузив его по адресу `https://bootstrap.pypa.io/get-pip.py`.
3. Запустите этот сценарий в версии Python 3 следующей командой:
   ```
   $ python3 get-pip.py
   ```
4. Установите распределенную систему контроля версий Git.
   Инструкции по ее загрузке и установке находятся по адресу `https://git-scm.com/downloads`.
5. Загрузите исходный код, прилагаемый к этой книге, выполнив следующие команды:
   ```
   $ git clone https://github.com/jimmysong/programmingbitcoin
   $ cd programmingbitcoin
   ```
6. Установите виртуальную среду `virtualenv` следующей командой:
   ```
   $ pip install virtualenv --user
   ```
7. Установите требования.
   В Linux/macOS — следующими командами:
   ```
   $ virtualenv -p python3 .venv
   $ . .venv/bin/activate
         (.venv) $ pip install -r requirements.txt
   ```
   В Windows — следующими командами:
   ```
   C:\programmingbitcoin> virtualenv -p
   C:\PathToYourPythonInstallation\Python.exe .venv
   C:\programmingbitcoin> .venv\Scripts\activate.bat
   C:\programmingbitcoin> pip install -r requirements.txt
   ```
8. Запустите командную оболочку Jupyter Notebook:
   ```
    (.venv) $ jupyter notebook
   [I 11:13:23.061 NotebookApp] Serving notebooks
     from local directory:[2]
       /home/jimmy/programmingbitcoin
   [I 11:13:23.061 NotebookApp]
     The Jupyter Notebook is running at:[3]
   ```

[2] Обслуживание блокнотов из следующего локального каталога:

[3] Jupyter Notebook выполняется по адресу:

```
[I 11:13:23.061 NotebookApp] http://localhost:8888/?token=
    f849627e4d9d07d2158e3fcde93590eff4a9a7a01f65a8e7
[I 11:13:23.061 NotebookApp]
  Use Control-C to stop this server and
  shut down all kernels (twice to skip confirmation).[4]
[C 11:13:23.065 NotebookApp]

      Copy/paste this URL into your browser when you connect
      for the first time, to login with a token:[5]
          http://localhost:8888/?token=
          f849627e4d9d07d2158e3fcde93590eff4a9a7a01f65a8e7
```

В итоге браузер должен открыться автоматически, как показано на рис. 1.

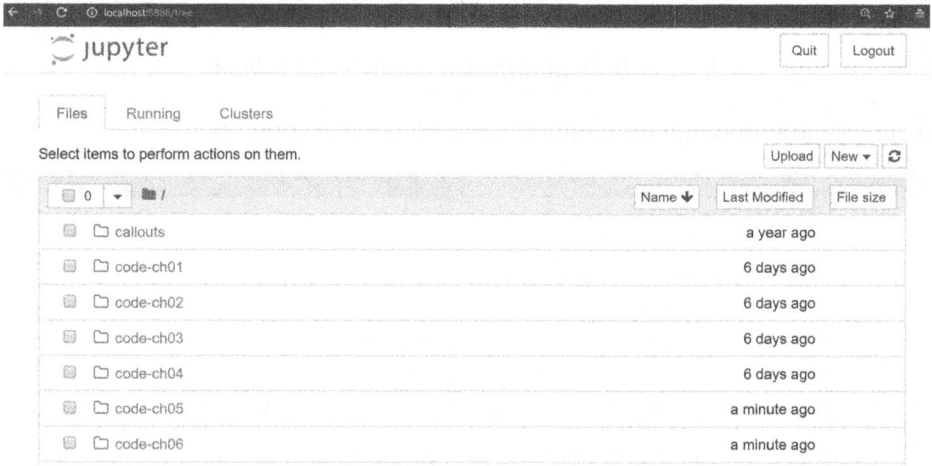

Рис. 1. Командная оболочка Jupyter Notebook

Отсюда можно перейти к каталогам, распределенным по отдельным главам книги. Так, для выполнения упражнений из главы 1 перейдите к каталогу code-ch01 (рис. 2).

[4] Чтобы остановить сервер и все ядра, нажмите комбинацию клавиш <Ctrl+C> (дважды, чтобы пропустить подтверждение).

[5] Скопируйте и вставьте этот URL в окне своего браузера при первом подключении, чтобы зарегистрироваться с указанным в нем токеном:

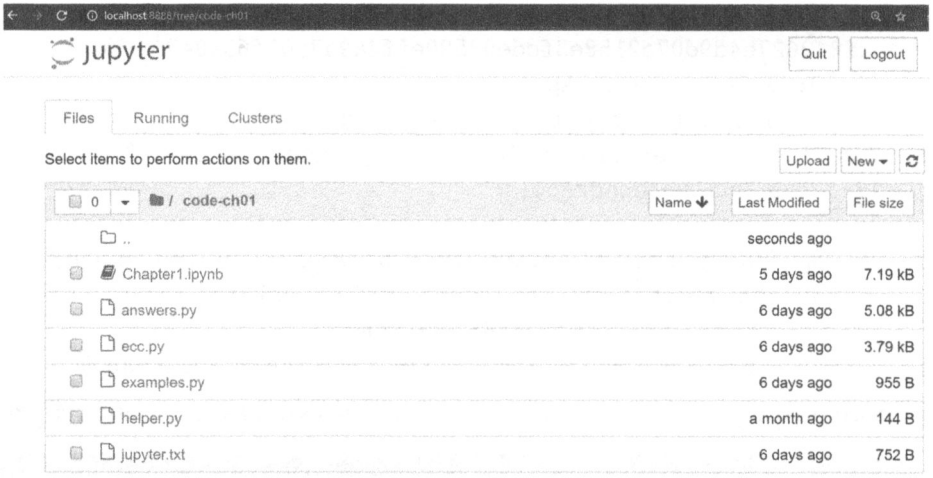

Рис. 2. Вид каталогов в Jupyter Notebook

А далее можно открыть файл `Chapter1.ipynb` (рис. 3).

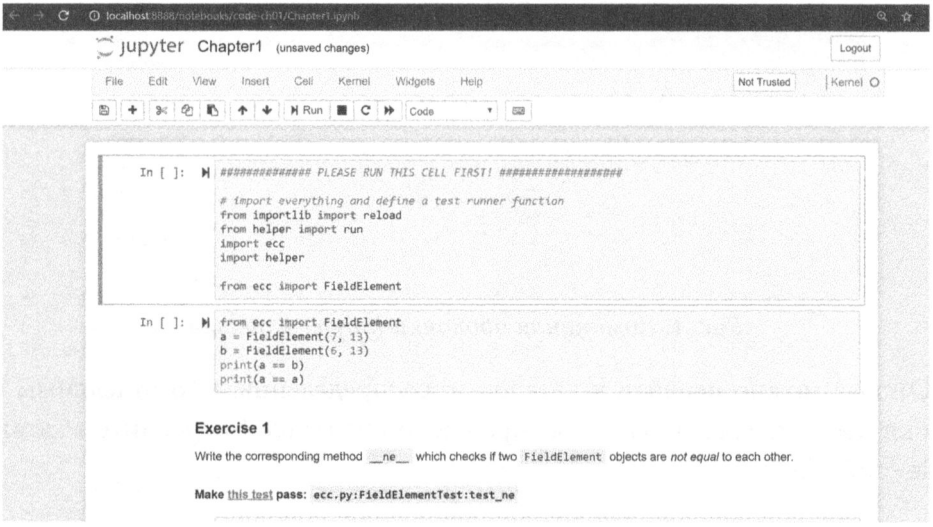

Рис. 3. Вид блокнота в Jupyter Notebook

Вам, возможно, придется освоиться с таким интерфейсом, если вы пользуетесь им впервые. Но истинный смысл командной оболочки Jupyter Notebook состоит в том, что в ней можно выполнять код Python из браузера, что упрощает экспериментирование с ним. Каждую "ячейку" кода можно выполнять по отдельности, чтобы наблюдать результаты, как будто это интерактивная оболочка Python.

Большая часть упражнений посвящена понятиям из области программирования, представленным в этой книге. И хотя модульные тесты уже написаны для вас, вам все же придется написать код Python для прохождения тестов. Проверить правильность написанного кода можно и непосредственно в Jupyter Notebook. Для этого необходимо отредактировать соответствующий файл, щелкнув на ссылке `this test` (данный тест; см. рис. 3). По этой ссылке произойдет переход на вкладку браузера, аналогичную приведенной на рис. 4. Отредактируйте здесь файл и сохраните его, чтобы тест был выполнен.

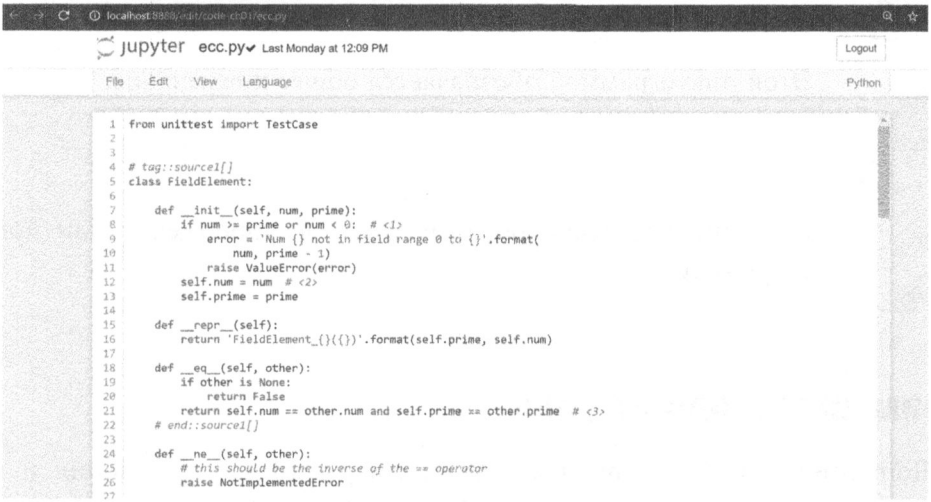

Рис. 4. Редактирование исходного файла `ecc.py`

Ответы к упражнениям

Все ответы к упражнениям, предлагаемым в этой книге, приведены в приложении. Они доступны также в файлах `code-ch<xx>/answers.py`, где *<xx>* — номер соответствующей главы.

Соглашения, принятые в этой книге

В этой книге приняты следующие условные обозначения.

Курсивом выделяются новые термины.

Моноширинным шрифтом выделяются адреса электронной почты, имена и расширения файлов, элементы кода (переменные, имена функций, операторы, переменные окружения, ключевые слова и типы данных), а также исходный код в листингах.

Полужирным моноширинным шрифтом выделяются команды или другой текст, который должен быть введен пользователем.

Наклонным моноширинным шрифтом выделяется текст, который должен быть заменен предоставляемыми пользователем значениями или же теми значениями, которые определяются по контексту.

Этой пиктограммой обозначается совет или рекомендация.

Этой пиктограммой обозначается общее примечание.

Этой пиктограммой обозначается предупреждение или предостережение.

Примеры исходного кода

Дополнительный материал книги (примеры кода, упражнения и пр.) доступен для загрузки по адресу `https://github.com/jimmysong/programmingbitcoin`. Эта книга служит учебным пособием, помогающим читателю решать стоящие перед ним задачи разработки прикладных программ. В общем, приведенные здесь примеры кода можно использовать в своих программах и документации. Для этого не нужно спрашивать разрешение у автора или издателя. Так, для применения в прикладной программе нескольких фрагментов кода из примеров в этой книге специальное разрешение не требуется. Но для продажи или распространения в иных целях на внешних носителях фрагментов кода из примеров обязательно нужно получить разрешение от издательства O'Reilly. Для цитирования текста и примеров кода из этой книги в ответах на вопросы специальное разрешение не требуется. Но для внедрения значительной части примеров кода в документацию на собственную продукцию обязательно необходимо разрешение от издательства O'Reilly.

Ссылки на эту книгу как на первоисточник желательны, но не обязательны. В ссылке обычно указываются название книги, автор, издатель и ISBN:

Благодарности

Тех, кого мне хотелось бы поблагодарить, — целый легион. Мне как автору этой книги пришлось черпать знания и опыт из многих источников, и поэтому трудно воздать должное всем, кто послужил такими источниками. Этим я хочу сказать, что если я забуду кого-нибудь упомянуть, то приношу им искреннее извинение.

Прежде всего, мне хотелось бы поблагодарить Господа Иисуса Христа, направившего меня на этот путь. Если бы не моя вера, я бы не взялся из этических убеждений вести блог о важности устойчивой валюты вообще и биткойна в частности, что в конечном итоге привело к написанию этой книги.

Мои родители Кэти и Кьюнг-Суп набрались мужества эмигрировать в Америку, когда мне было 8 лет от роду, что в конечном счете привело к тем возможностям, которые у меня появились. Мой отец подарил мне первые мои компьютеры (Commodore 16, ПК Hyundai на обыкновенном процессоре 8086 и еще один ПК на процессоре 486 с тактовой частотой 33 МГц от производителя, название которого я позабыл). А когда я учился в шестом и седьмом классах, моя мать наняла для меня частную преподавательницу по программированию, имя которой я уже не помню. Даже не знаю, как и где она ее нашла, когда я стал проявлять склонность к программированию. Эта позабытая мной преподавательница поддержала мое стремление к программированию, и, я надеюсь, моя признательность найдет свой путь к ней.

На своем жизненном пути я встретил немало учителей, и некоторых из них я в свое время просто ненавидел. В частности, мистер Марэн, миссис Эдельман и миссис Нельсон учили меня математике и вычислительной технике еще в школе. И хотя я совсем не являюсь сторонником современной образовательной системы, то, чему меня научили в школе, поспособствовало зарождению во мне любви к математике и программированию.

Мой одноклассник Эрик Сильберштейн дал мне после колледжа первую работу программистом в компании Idiom Technologies. Я собирался было

пойти по пути консультанта, когда судьбоносный телефонный звонок в 1998 году повел меня по пути программиста, которого я, по существу, с тех пор и не покидал.

На своей первой работе я познакомился с Кеном Лю, и просто удивительно, как мы занялись написанием книг, каждый на своем уровне. Он не только дал мне дельный совет по поводу издания книги, но и написал чудесное предисловие к ней. Кроме того, он отзывчивый и изумительный товарищ, и знакомство с ним для меня — большая честь.

Что же касается биткойна, то анонимный разработчик Сатоши Накамото изобрел то, что прежде казалось невозможным: децентрализованный цифровой дефицит. Изобретение биткойна имеет принципиальное значение, которое мир еще не до конца осознал. Я познакомился с биткойном на веб-сайте Slashdot (`https://slashdot.org/`) в 2011 году, а первую свою работу в качестве разработчика биткойна я получил от Алекса Мизрахи в 2013 году. И тогда я даже не представлял, что я должен был делать, хотя и немало узнал о биткойне под его руководством.

В 2013 году я встретил немало биткойнеров в группе Austin Bitcoin Meetup, а на следующей конференции Texas Bitcoin Conference в 2014 году — огромное количество людей, которых я знаю и поныне. И хотя я больше не принимаю участия не только в этой группе и конференции, я весьма признателен тем, кого я там встретил, в частности — Майклу Гольдштейну, Дениэлу Кравишу и Наполеону Коулу.

Алан Рейнер нанял меня на работу над проектом Armory еще в 2014 году, и я благодарен ему за возможность участвовать в таком значительном проекте.

Затем я работал в компании Paxos/itBit, где исполнительным директором был Чэд Каскарилла, а вице-президентом по проектированию — Радж Наир, которые расширили мой кругозор, пока я там работал. В частности, Радж обязал меня делать записи в блоге от компании Paxos/itBit, что, как ни удивительно, мне понравилось. И впоследствии я стал писать блог-посты на своем веб-сайте, что в конечном счете привело к организации семинаров и написанию этой книги.

Особого упоминания заслуживают трое моих коллег, с которыми я познакомился в компании Paxos/itBit. В частности, Ричард Кисс, создатель приложения `pycoin`, побудил меня к написанию этой книги, предложив писать ее вместе. И хотя на каком-то этапе мне пришлось писать ее самому, я все же благодарен ему за эту идею. Аарон Кезуэлл, будучи замечательным разработчиком, помог мне в организации семинаров и рецензировании этой книги.

Он не только прекрасный программист и математик, но, как я слышал, прилично владеет приемами каратэ. Майкл Флэксмен рецензировал все, что я написал о биткойне, включая мой блог, многие из моих библиотек в информационном хранилище GitHub и эту книгу. Он также помог мне в организации семинаров; и он — просто хороший собеседник. Майкл из тех, кто делает других лучше, и я благодарен ему за дружбу, которая для меня много значит.

В 2017 году приложение Vortex, Томас Хант и Тоун Вэйз привели меня на канал World Crypto Network, где началась моя деятельность на канале YouTube. В частности, Тоун вдохновлял меня на тяжкие труды и преданность делу.

Джон Ньюбери оказал мне помощь, когда я впервые принял участие в проекте Bitcoin Core; и он — просто хороший человек. Джон стал заметным участником данного проекта за относительно короткое время, что свидетельствует о его даровании и преданности делу. Благодарю также других участников проекта Bitcoin Core, в том числе Марко Фальке, Владимира ван дер Лаана, Алекса Моркоса, Пьетера Вуилле, Мэтта Коралло, Сухаза Дафтуара и Грега Максуэлла, которые просматривали написанный мною код и записи в блоге.

Особой благодарности заслуживает Дэвид Хардинг за научное рецензирование этой книги. Он трижды просмотрел всю ее рукопись, сделав немало полезных комментариев. Ему бы лучше написать когда-нибудь книгу о биткойне, поскольку он обладает энциклопедическими знаниями практически обо всем, что произошло в истории биткойна.

Джим Кельвин помог мне связаться с издательством O'Reilly, а Майк Лукидес как приемный редактор дал зеленый свет данному проекту. Андреас Антонополус дал мне ряд ценных указаний и рекомендовал меня издательству O'Reilly. Мишель Кронин помогала мне придерживаться намеченного графика, когда я целый год был полностью поглощен написанием этой книги. Кристен Браун как производственный редактор сделала немало для того, чтобы эта книга вовремя увидела свет, а Джеймс Фрелай выполнил ее литературное редактирование. Я большой поклонник издателя Тима О'Райли, который сослужил немалую службу инженерному сообществу, выпуская такие замечательные книги.

В процессе написания этой книги мне помогала целая компания биткойнеров из группы Austin Bitcoin Meetup, в том числе Брайан Бишоп, Уилл Коул, Наполеон Коул, Типтон Коул, Тур Демеестер, Джонни Дилли, Майкл Флэксмен, Паркер Льюис, Джастин Мун, Алан Пискителло и Эндрю Поельстра. Следует также отметить канал Slack (от компании TAAS), на котором Сайфедин Аммус заявил, что он пишет книгу, успех которой вдохновил и меня.

Помимо биткойнеров из GitHub, бывшие слушатели моего курса по программированию биткойна также приняли участие в рецензировании этой книги. В частности, Джефф Флауэрс, Брайан Лиотти, Кэйзи Боумен, Джонсон Лау, Альберт Чен, Джейсон Лез, Томас Браунбергер, Эдуардо Кобейн и Спенсер Хенсон выявили ряд недостатков в рукописи. А мой преданный помощник Катрина Хавьер помогала мне в составлении многих диаграмм к этой книге.

Мои подписчики на канале YouTube, последователи в Twitter и читатели в Medium помогли мне найти себя и стать на путь предпринимательства.

И наконец, моя жена Джули и мои дети оказывали мне всяческую поддержку в течение двух последних лет. Если бы не они, то сомневаюсь, что у меня были бы стимулы так много работать.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`

WWW: `http://www.dialektika.com`

Конечные поля

Самое сложное в программировании биткойна — выяснить, с чего следует начать. Ведь существует столько взаимозависимых компонентов, что изучение одного из них может привести к необходимости изучить другой, а тот, в свою очередь, — изучить еще что-нибудь, прежде чем разобраться в первоначальном компоненте.

Назначение этой главы — облегчить вам начало. Как ни странно, мы начнем с основ математики, которые потребуются, чтобы уяснить особенности криптографии на эллиптических кривых. А криптография на эллиптических кривых, в свою очередь, даст представление об алгоритмах подписания и верификации. Ведь именно на этих алгоритмах основывается принцип действия транзакций, которые являются атомарными единицами перевода стоимости в биткойне. Изучив сначала конечные поля и эллиптические кривые, вы сможете твердо усвоить понятия, необходимые для логичного продвижения вперед.

Но имейте в виду, что материал этой и двух последующих глав сродни вегетарианской диете, особенно если вы давно не занимались формальной математикой. Тем не менее прочтите их, поскольку представленные в них понятия и примеры кода будут использоваться в остальной части книги.

Высшая математика

Изучение математических структур может оказаться несколько пугающим занятием, но можно надеяться, что в этой главе миф о трудностях освоения высшей математики будет развеян. В частности, для изучения конечных полей достаточно математических знаний на уровне алгебры.

Конечные поля можно рассматривать как математическую дисциплину, вполне приемлемую для изучения вместо тригонометрии, если бы в образовательной системе не было принято решение, что изучать тригонометрию

важнее. Этим я хочу сказать, что изучить конечные поля нетрудно и для этого потребуется не больше предварительной подготовки, чем знание алгебры.

В этой главе вам нужно будет разобраться в криптографии на эллиптических кривых. А криптография на эллиптических кривых требуется для того, чтобы уяснить алгоритмы подписания и верификации, составляющие саму суть биткойна. Как упоминалось выше, материал этой и двух последующих глав может показаться, на первый взгляд, никак не связанным с рассматриваемым здесь предметом; тем не менее настоятельно рекомендуется их прочитать. Основы, изложенные в этих главах, упростят понимание не только биткойна, но и подписей Шнорра, конфиденциальных транзакций и прочих передовых технологий биткойна.

Определение конечного поля

В математике *конечное поле* (finite field) определяется как конечное множество чисел, над которыми можно выполнить две математические операции, + (сложения) и × (умножения), удовлетворяющие перечисленным ниже условиям.

1. Если a и b принадлежат множеству, то результаты операций $a + b$ и $a \times b$ также принадлежит этому множеству. Такое свойство можно назвать *замкнутым* (closed).
2. Нуль существует и обладает свойством $a + 0 = a$. Такое свойство можно назвать *аддитивным тождеством* (additive identity).
3. Единица существует и обладает свойством $a \times 1 = a$. Такое свойство можно назвать *мультипликативным тождеством* (multiplicative identity).
4. Если a принадлежит множеству, то ему же принадлежит и $-a$, определяемое как значение, удовлетворяющее выражению $a + (-a) = 0$. Такое свойство можно назвать *аддитивной инверсией* (additive inverse).
5. Если a не равно 0 и принадлежит множеству, то ему же принадлежит и a^{-1}, определяемое как значение, удовлетворяющее выражению $a \times a^{-1} = 1$. Такое свойство можно назвать *мультипликативной инверсией* (multiplicative inverse).

Рассмотрим каждое из перечисленных выше условий более подробно.

Мы работаем с конечным множеством чисел. Поскольку оно конечное, то можно задать некоторое число p, определяющее размер множества, называемое также *порядком* (order) множества.

Так, условие №1 говорит о том, что множество замкнуто относительно сложения и умножения. Это означает, что операции сложения и умножения определяются таким образом, чтобы их результаты принадлежали данному множеству. Например, множество {0,1,2} *не* является замкнутым относительно сложения, поскольку 1 + 2 = 3 и 2 + 2 = 4, но ни число 3, ни число 4 не принадлежит данному множеству. Безусловно, операцию сложения можно определить несколько иначе, чтобы сделать данное множество замкнутым, но с помощью "обычной" операции сложения такое множество нельзя сделать замкнутым. А с другой стороны, множество {–1,0,1} является замкнутым относительно умножения. Если перемножить два любых числа из этого множества, а всего таких комбинаций девять, то их произведение всегда будет принадлежать множеству.

В математике имеется и другая возможность определить операцию умножения особым образом, чтобы сделать подобные множества замкнутыми. О том, как именно определить операции сложения и умножения, речь пойдет далее в этой главе, а до тех пор следует сказать, что операции сложения и вычитания можно определить *иначе*, чем вам, вероятно, известно.

В частности, условия №2 и 3 определяют аддитивное и мультипликативное тождества. Это означает, что числа 0 и 1 принадлежат множеству.

Условие №4 определяет аддитивную инверсию. Это означает, что если a принадлежит множеству, то ему же принадлежит и $-a$. Используя аддитивную инверсию, можно определить операцию вычитания.

Далее, условие №5 определяет, что аналогичным свойством обладает и операция умножения. Если a принадлежит множеству, то ему же принадлежит и a^{-1}. Это означает, что $a \times a^{-1} = 1$. Используя мультипликативную инверсию, можно определить операцию деления. Следует заметить, что для конечного поля это сделать будет непросто.

Определение конечных множеств

Если порядок (или размер) множества равен p, то элементы множества можно обозначить как 0, 1, 2, ... p–1. Именно эти числа, а совсем не обязательно традиционные числа 0, 1, 2, 3 и так далее, называются *элементами* множества. И хотя они во многом ведут себя, как традиционные числа, они

различаются порядком выполнения операций сложения, вычитания, умножения и пр.

В математическом обозначении множество конечных полей выглядит следующим образом:

$$F_p = \{0, 1, 2, \ldots p-1\}.$$

Каковы же элементы множества конечных полей? F_p — это конкретное конечное поле, называемое "полем p", "полем 29" или полем любого размера, которое в математике называется *порядком*. А числа в фигурных скобках {} обозначают элементы, находящиеся в полях. Эти элементы именуются как 0, 1, 2 и так далее поскольку так удобней для наших целей.

Так, конечное поле порядка 11 выглядит следующим образом:

$$F_{11} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

А конечное поле порядка 17 — таким образом:

$$F_{17} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}.$$

И наконец, конечное поле порядка 983 выглядит так, как показано ниже.

$$F_{983} = \{0, 1, 2, \ldots 982\}$$

Следует, однако, иметь в виду, что порядок конечного поля всегда на 1 больше значения его последнего элемента. Вы, вероятно, обратили внимание на то, что порядок конечного поля всякий раз оказывается равным простому числу. По разным причинам, которые станут понятны в дальнейшем, оказывается, что порядок конечных полей *должен* быть равен степени простого числа и что наибольший для нас интерес представляют именно те конечные поля, порядок которых равен простому числу.

Построение конечного поля на языке Python

Нам требуется каким-то образом представить каждый элемент конечного поля. Поэтому создадим в Python класс, представляющий единственный элемент конечного поля. Этому классу естественно присвоить имя `FieldElement`. Этот класс представляет элемент из конечного поля F_{prime}. Скелетное представление этого класса выглядит следующим образом:

```
class FieldElement:

  def __init__(self, num, prime):
    if num >= prime or num < 0:     ❶
      error = 'Num {} not in field range 0 to {}'.format(
```

```
            num, prime - 1)
        raise ValueError(error)
    self.num = num      ❷
    self.prime = prime

    def __repr__(self):
        return 'FieldElement_{}({})'.format(
            self.prime, self.num)

    def __eq__(self, other):
        if other is None:
            return False
        return self.num == other.num
                and self.prime == other.prime      ❸
```

❶ Сначала в классе проверяется, находится ли значение свойства num в пределах от 0 до prime-1 включительно. Если это не так, то элемент конечного поля типа FieldElement некорректен и передается исключение типа ValueError, которое должно быть непременно передано, если получено неподходящее значение.

❷ В остальной части метода __init__() создаваемому объекту данного класса присваиваются инициализирующие значения.

❸ В методе проверяется равенство двух объектов класса FieldElement. И эта проверка дает истинный результат лишь в том случае, если свойства num и prime равны.

То, что мы уже определили, позволяет сделать следующее:

```
>>> from ecc import FieldElement
>>> a = FieldElement(7, 13)
>>> b = FieldElement(6, 13)
>>> print(a == b)
False
>>> print(a == a)
True
```

В языке Python можно заменить операцию == методом __eq__(), что и было сделано в классе FieldElement и чем можно выгодно воспользоваться в дальнейшем, как демонстрируется в примерах кода, прилагаемых к этой книге. Установив командную оболочку Jupyter Notebook (см. раздел "Подготовка" во введении этой книги), можно перейти к файлу code-ch01/Chapter1.ipynb и выполнить находящийся в нем код, чтобы увидеть полученные результаты. А для выполнения следующего упражнения откройте

файл `ecc.py`, щелкнув на ссылке в блоке Exercise 1 (Упражнение 1). Если у вас возникнут затруднения при выполнении данного упражнения, помните, что ответ к каждому упражнению этой книги, можно найти в Приложении.

Упражнение 1

Напишите соответствующий метод `__ne__()`, в котором оба объекта типа `FieldElement` проверяются на *неравенство*.

Арифметика по модулю

Одним из средств, позволяющих сделать конечное поле замкнутым относительно сложения, вычитания, умножения и деления, является *арифметика по модулю* (modulo arithmetic).

Используя арифметику по модулю, можно определить операцию сложения в конечном множестве, что вам должно быть известно с тех пор, как вы научились делить в столбик. Вспомните, как это делается, глядя на пример задачи, приведенной на рис. 1.1.

$$3\overline{)7}\quad 2R1$$

Рис. 1.1. Первый пример деления в столбик

Всякий раз, когда не происходит деления нацело, возникает "остаток", т.е. величина, остающаяся от фактического деления одного числа на другое[1]. Аналогичным образом определяется и модуль, а операция по модулю обозначается символом `%`:

```
7 % 3 = 1
```

Еще один пример деления в столбик приведен на рис. 1.2.

$$7\overline{)27}\quad 3R6$$

Рис. 1.2. Второй пример деления в столбик

[1] На рис. 1.1 R1 обозначает остаток от деления 7 и 3, равный 1. — *Примеч. ред.*

Рассуждая формально, операция по модулю означает получение остатка от деления одного числа на другое. Ниже приведен еще один пример такой операции над большими числами.

1747 % 241 = 60

Если угодно, арифметику по модулю можно рассматривать как арифметику "оборота стрелок часов". Рассмотрим в качестве примера следующую задачу.

Сейчас 3 часа. Что будет показывать часовая стрелка через 47 часов?

Решение: 2 часа, поскольку (3 + 47) % 12 = 2 (рис. 1.3).

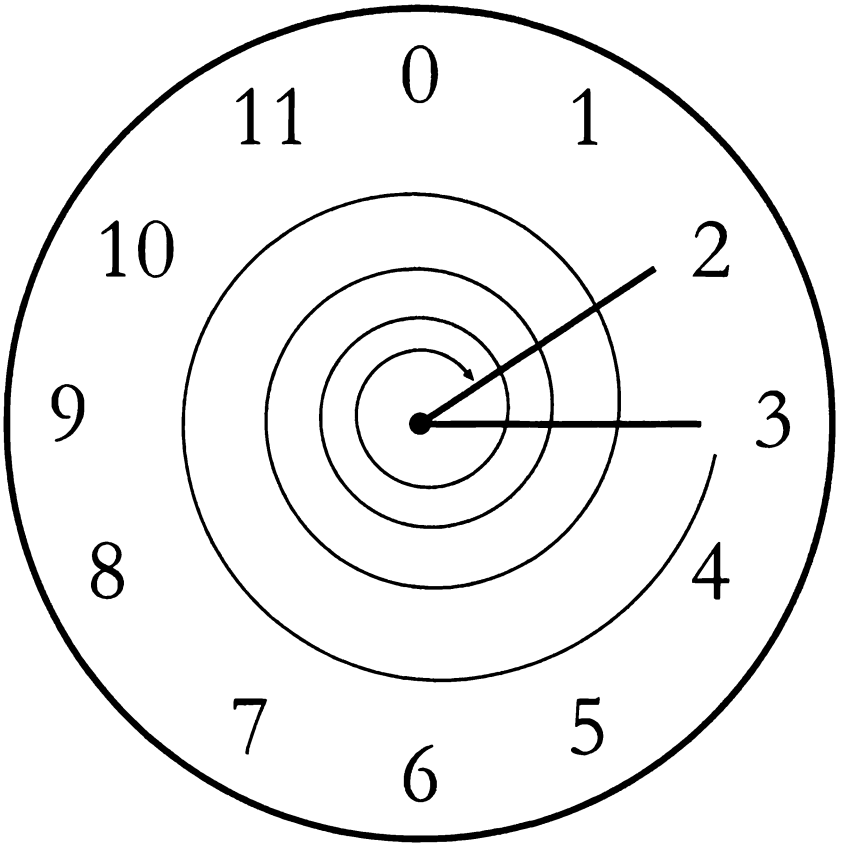

Рис. 1.3. Перемещение часовой стрелки на 47 часов вперед

Такой "оборот стрелок" можно рассматривать и в смысле перехода через 0 всякий раз, когда часовая стрелка перемещается на 12 часов вперед.

Операцию по модулю можно выполнять и над отрицательными числами. В качестве еще одного примера рассмотрим такую задачу.

Сейчас 3 часа. Что показывала часовая стрелка 16 часов назад?

Решение: 11 часов.

$$(3 - 16) \% 12 = 11$$

Перемещение минутной стрелки часов также является операцией по модулю. В качестве примера можно привести следующую задачу.

Сейчас 12 минут второго. Что будет показывать минутная стрелка через 843 минуты?

Решение: она будет показывать 15 минут второго.

$$(12 + 843) \% 60 = 15$$

Аналогично можно сформулировать такую задачу.

Сейчас 23 минуты второго. Что будет показывать минутная стрелка через 97 минут?

В данном случае минутная стрелка будет показывать 0 минут.

$$(23 + 97) \% 60 = 0$$

0 — это еще одно средство для обозначения отсутствия остатка.

Результат операции по модулю (%) для минут всегда находится в пределах от 0 до 59 включительно. И это свойство оказывается очень удобным, поскольку даже очень большие числа можно свести к относительно малым пределам с помощью операции по модулю, как показано ниже.

$$14738495684013 \% 60 = 33$$

Мы будем пользоваться далее операцией по модулю, определяя арифметику конечных полей. В большинстве операций над конечными полями операция по модулю применяется в некоторой форме.

Арифметика по модулю на языке Python

В языке Python для арифметики по модулю служит операция `%`. Ниже показано, каким образом она применяется непосредственно в коде.

```
>>> print(7 % 3)
1
```

Операцию по модулю можно выполнять и над отрицательными числами, как показано ниже.

```
>>> print(-27 % 13)
12
```

Сложение и вычитание конечных полей

Напомним, что операцию сложения в конечном поле необходимо определить таким образом, чтобы ее результат гарантированно остался в множестве. Таким образом, требуется обеспечить *замкнутость* операции сложения в конечном поле.

Используя рассмотренную выше арифметику по модулю, можно сделать операцию сложения замкнутой. Допустим, что имеется конечное поле 19:

$$F_{19} = \{0, 1, 2, \ldots 18\}$$

Здесь $a, b \in F_{19}$. Обратите внимание на то, что $\in$ означает "является элементом". В данном случае a и b являются элементами конечного поля F_{19}.

Замкнутость операции сложения обозначается следующим образом:

$$a +_f b \in F_{19}$$

Операция сложения в конечном поле обозначается как $+_f$ во избежание путаницы с обычной целочисленной операцией сложения, обозначаемой как +.

Если воспользоваться арифметикой по модулю, то можно гарантировать замкнутость операции сложения в конечном поле. В частности, операцию $a+_f b$ можно определить следующим образом:

$$a +_f b = (a + b) \% 19$$

Например:

$$7 +_f 8 = (7 + 8) \% 19 = 15$$
$$11 +_f 17 = (11 + 17) \% 19 = 9$$

И т.д.

В замкнутой операции сложения из множества берутся два числа, которые складываются, а конечный результат "оборачивается", чтобы получить сумму. В данном случае мы создаем свою операцию сложения, хотя ее результат не вполне очевиден. Выражение $11+_f 17 = 19$ выглядит неверно, поскольку мы просто не привыкли к операции сложения в конечном поле.

В общем случае операция сложения в конечном поле определяется следующим образом:

$$a +_f b = (a + b) \% p$$

где $a, b \in F_p$.

Аддитивная инверсия определяется так, как показано ниже, где из $a \in F_p$ следует, что $-a \in F_p$.

$$-_f a = (-a) \% p$$

И снова для ясности обозначение $-_f$ помогает отличать операций вычитания и инверсии в конечном поле от обычных целочисленных операций вычитания и инверсии.

Так, операция инверсии в конечном поле F_{19}

$$-_f 9 = (-9) \% 19 = 10$$

означает, что

$$9 +_f 10 = 0$$

и оказывается истинной.

Аналогично можно выполнить операцию вычитания в конечном поле:

$a -_f b = (a - b) \% p$

где $a, b \in F_p$.

Так, в конечном поле F_{19} можно выполнить следующие операции вычитания:

$11 -_f 9 = (11-9) \% 19 = 2$
$6 -_f 13 = (6-13) \% 19 = 12$
И т.д.

Упражнение 2

Решите приведенные ниже задачи в конечном поле F_{57} при условии, что все знаки + здесь обозначают операцию сложения $+_f$, а все знаки - — операцию вычитания $-_f$.

- 44 + 33
- 9 - 29
- 17 + 42 + 49
- 52 - 30 - 38

Программная реализация операций сложения и вычитания на языке Python

Теперь в классе `FieldElement` можно определить методы `__add__()` и `__sub__()` для реализации операций сложения и вычитания в конечном поле. Суть этих методов в том, что в них можно выполнять операции, аналогичные приведенным ниже.

```
>>> from ecc import FieldElement
>>> a = FieldElement(7, 13)
>>> b = FieldElement(12, 13)
>>> c = FieldElement(6, 13)
>>> print(a+b==c)
True
```

В методе `__add__()` данного класса можно определить, что именно обозначает операция сложения (+). Как это сделать на языке Python? Для этого воспользуемся приобретенными знаниями арифметики по модулю и создадим новый метод в классе `FieldElement`, как показано ниже.

```
def __add__(self, other):
  if self.prime != other.prime:     ❶
    raise TypeError('Cannot add two numbers in different Fields')
  num = (self.num + other.num) % self.prime     ❷
  return self.__class__(num, self.prime)     ❸
```

❶ Мы должны убедиться, что элементы принадлежат к одному и тому же конечному полю, а иначе данное вычисление не имеет никакого смысла.

❷ Операция сложения в конечном поле определяется с помощью операции по модулю, как пояснялось ранее.

❸ Мы должны возвратить экземпляр класса, к которому удобно получать доступ с помощью метода `self.__class__()`. Для этого данному методу передаются два инициализирующих аргумента, `num` и `self.prime`, как и для упоминавшегося ранее метода `__init__()`.

Следует заметить, что вместо метода `self.__class__()` можно было бы просто воспользоваться классом `FieldElement`, но тогда сделать этот метод наследуемым было бы непросто. Подклассификацию класса `FieldElement` мы еще осуществим в дальнейшем, а здесь нам важно сделать метод наследуемым.

Упражнение 3

Напишите соответствующий метод `__sub__()`, в котором определяется операция вычитания двух объектов типа `FieldElement`.

Операции умножения и возведения в степень в конечном поле

Подобно тому, как была ранее определена новая операция сложения ($+_f$) в конечном поле, можно определить новую замкнутую операцию умножения

для конечных полей. А многократно умножая одно и то же число, можно также определить новую операцию возведения в степень. И в этом разделе поясняется, как определить подобные операции с помощью арифметики по модулю.

Операция умножения, по существу, означает многократное сложение чисел, как демонстрируется в следующих примерах:

$5 \times 3 = 5 + 5 + 5 = 15$

$8 \times 17 = 8 + 8 + 8 + \ldots$ (всего 17 чисел 8) $\ldots + 8 = 136$

Аналогичным образом можно определить операцию умножения в конечном поле. Так, оперируя снова конечным полем F_{19}, операции умножения в нем можно обозначить следующим образом:

$5 \times_f 3 = 5 +_f 5 +_f 5$

$8 \times_f 17 = 8 +_f 8 +_f 8 +_f \ldots$ (всего 17 чисел 8) $\ldots +_f 8$

Как вам должно быть уже известно, в правой части приведенных выше выражений получается число, принадлежащие конечному множеству F_{19}:

$5 \times_f 3 = 5 +_f 5 +_f 5 = 15 \,\%\, 19 = 15$

$8 \times_f 17 = 8 +_f 8 +_f 8 + \ldots$ (всего 17 чисел 8) $\ldots +_f 8 =$

$(8 \times 17) \,\%\, 19 = 136 \,\%\, 19 = 3$

Обратите внимание на то, что результат второй операции умножения не вполне очевиден. Мы обычно не считаем, что $8 \times_f 17 = 3$, но это именно та необходимая часть операции умножения в конечном поле, без которой ее нельзя определить как замкнутую. Это означает, что результат операции умножения в конечном поле всегда принадлежит множеству $\{0, 1, \ldots p-1\}$.

А операцию возведения в степень можно просто определить как многократное умножение числа:

$7^3 = 7 \times_f 7 \times_f 7 = 343$

В конечном поле операцию возведения в степень можно выполнить, используя арифметику по модулю, как показано ниже на примере конечного поля F_{19}.

$7^3 = 343 \,\%\, 19 = 1$

$9^{12} = 7$

Операция возведения в степень также дает не вполне очевидные результаты, ведь обычно мы не считаем, что $7^3 = 1$ или $9^{12} = 7$. Но, опять же, конечные поля должны быть определены таким образом, чтобы выполнение

арифметических операций в них *всегда* приводило в конечном итоге к числу, находящемуся в данном поле.

Упражнение 4

Решите приведенные ниже уравнения в конечном поле F_{97} при условии, что операции умножения и возведения в степень выполняются в данном конечном поле.

- $95 \times 45 \times 31$
- $17 \times 13 \times 19 \times 44$
- $12^{7} \times 77^{49}$

Упражнение 5

Какое из приведенных ниже множеств чисел находится в конечном поле F_{19}, если $k = 1, 3, 7, 13, 18$? Есть ли в этих множествах что-нибудь примечательное?

$$\{k \times 0, k \times 1, k \times 2, k \times 3, \ldots k \times 18\}$$

Почему поля оказываются простыми

В ответе к упражнению 5 объясняется, почему поля должны содержать количество элементов, составляющее степень *простого* числа. Неважно, какое значение k выбрано, если оно больше нуля, то умножение всего множества на k даст в конечном итоге то же самое множество, что и первоначальное.

Очевидно, что порядок простого числа приводит к равнозначности каждого элемента конечного поля. Если бы порядком множества было составное число, то умножение данного множества на один из делителей привело бы к меньшему множеству.

Программная реализация операции умножения на языке Python

Теперь, когда стало ясно, какой должна быть операция умножения в конечном поле, в классе `FieldElement` можно определить метод `__mul__()`, заменяющий обычную операцию умножения `*`. Такая операция должна выполняться следующим образом:

```
>>> from ecc import FieldElement
>>> a = FieldElement(3, 13)
>>> b = FieldElement(12, 13)
>>> c = FieldElement(10, 13)
>>> print(a*b==c)
True
```

В следующем упражнении предлагается определить в классе `FieldElement` метод `__mul__()`, реализующий операцию умножения в конечном поле аналогично операциям сложения и вычитания.

Упражнение 6

Напишите соответствующий метод `__mul__()`, в котором определяется операция умножения двух элементов конечного поля.

Программная реализация операции возведения в степень на языке Python

В классе `FieldElement` требуется определить метод `__pow__()`, заменяющий обычную операцию `**` языка Python. Отличие здесь в том, что показатель степени не относится к типу `FieldElement`, а потому требует несколько иного обращения. Такая операция должна выполняться следующим образом:

```
>>> from ecc import FieldElement
>>> a = FieldElement(3, 13)
>>> b = FieldElement(1, 13)
>>> print(a**3==b)
True
```

Поскольку показатель степени является целочисленным, вместо еще одного экземпляра класса `FieldElement` метод `__pow__()` должен принимать целочисленную переменную `exponent`. Ниже показано, каким образом данный метод определяется непосредственно в коде.

```
class FieldElement:
...
  def __pow__(self, exponent):
    num = (self.num ** exponent) % self.prime     ❶
    return self.__class__(num, self.prime)     ❷
```

❶ И хотя можно поступить и так, вызов метода `pow(self.num, exponent, self.prime)` эффективнее.

❷ Как и прежде, необходимо возвратить экземпляр данного класса.

А почему бы не сделать показатель степени объектом типа `FieldElement`? Оказывается, что показатель степени совсем не обязательно должен быть элементом конечного поля для выполнения математических операций. В действительности, если бы это было именно так, показатели степени не проявляли бы интуитивно ожидаемое от них поведение, как, например, сложение показателей степени при умножении по одному и тому же основанию.

Некоторые из реализуемых здесь алгоритмов могут оказаться медленными при обработке крупных чисел. Но мы воспользуемся в дальнейшем более искусными приемами, чтобы повысить производительность этих алгоритмов.

Упражнение 7

Если $p = 7, 11, 17, 31$, то какое из приведенных ниже множеств окажется в конечном поле F_p?

$\{1^{(p-1)}, 2^{(p-1)}, 3^{(p-1)}, 4^{(p-1)}, \ldots (p-1)^{(p-1)}\}$

Операция деления в конечном поле

Интуиция, помогающая нам определить операции сложения, вычитания, умножения и даже возведения в степень, к сожалению, не способна в такой же степени помочь при определении операции вычитания в конечном поле. Это самая сложная операция, и поэтому начнем с того, что должно быть более понятно.

В обычной арифметике деление является операцией обратной умножению, как демонстрируется в следующем примере:

из $7 \times 8 = 56$ следует, что $56/8 = 7$
из $12 \times 2 = 24$ следует, что $24/12 = 2$

И так далее. Данным обстоятельством можно воспользоваться как вспомогательным средством при определении операции деления в конечном поле. Следует, однако, иметь в виду, что делить на нуль, как и в обычной арифметике, нельзя.

В конечном поле F_{19} известно, что

из $3 \times_f 7 = 21 \ \% \ 19 = 2$ следует, что $2/_f 7 = 3$;
из $9 \times_f 5 = 45 \ \% \ 19 = 7$ следует, что $7/_f 5 = 9$.

И это вполне очевидно, поскольку выражения $2/_f 7$ и $7/_f 5$ обычно считаются дробями, а не естественными элементами конечного поля. Но ведь конечные

поля тем и примечательны, что они *замкнуты* относительно деления. Это означает, что если разделить одно число на другое при условии, что знаменатель не равен нулю, то в конечном итоге получится другой элемент конечного поля.

В связи с этим может возникнуть следующий вопрос: как вычислить выражение $2/_f 7$, если заранее неизвестно, что $3 \times_f 7 = 2$? Это, конечно, вполне резонный вопрос, и чтобы ответить на него, придется воспользоваться результатом, полученным из упражнения 7.

На тот случай, если вы не выполнили это упражнение, его решение в том, что $n^{(p-1)}$ всегда равно 1 для каждого простого числа p и $n > 0$. И этот прекрасный результат следует из теории чисел и, в частности, из так называемой малой теоремы Ферма, которая, по существу, гласит

$$n^{(p-1)} \% p = 1,$$

где p — простое число.

Малая теорема Ферма

Имеется немало доказательств этой теоремы, но самое простое из них, вероятно, в том, что приведенные ниже множества равны, как следует из упражнения 5.

$\{1, 2, 3, \ldots p-2, p-1\} = \{n \% p, 2n \% p, 3n \% p \; (p-2)n \% p, (p-1)n \% p\}$

Получаемые в итоге числа могут и не следовать в правильном порядке, но в обоих множествах оказываются одни и те же числа. В таком случае можно перемножить каждый элемент в обоих множествах, чтобы получить в итоге следующее равенство:

$1 \times 2 \times 3 \times \ldots \times (p-2) \times (p-1) \% p = n \times 2n \times 3n \times \ldots \times (p-2)n \times (p-1)n \% p$

Левую часть этого равенства можно свести к выражению $(p-1)! \% p$, где ! — факториал (например, $5! = 5 \times 4 \times 3 \times 2 \times 1$). А в правой его части можно собрать все члены n и получить в итоге следующее выражение:

$(p-1)! \times n^{(p-1)} \% p$

Таким образом,

$(p-1)! \% p = (p-1)! \times n^{(p-1)} \% p.$

Исключив член $(p-1)!$ В обеих частях этого равенства можно прийти к следующему равенству:

$1 = n^{(p-1)} \% p$

Что и служит доказательством малой теоремы Ферма.

И такой результат будет всегда истинным, поскольку мы оперируем в пределах простых полей.

А поскольку операция деления является обратной операции умножения, известно следующее:

$$a/b = a \times_f (1/b) = a \times_f b^{-1}$$

Задачу деления можно свести к задаче умножения при условии, что можно вычислить, чему равно b^{-1}. И здесь вступает в действие малая теорема Ферма. Известно, что

$$b^{(p-1)} = 1,$$

поскольку p — простое число. Таким образом,

$$b^{-1} = b^{-1} \times_f 1 = b^{-1} \times_f b^{(p-1)} = b^{(p-2)}$$

или

$$b^{-1} = b^{(p-2)}$$

В конечном поле F_{19} это практически означает, что $b^{18} = 1$, т.е. $b^{-1} = b^{17}$ для всех значений $b > 0$.

Иными словами, используя операцию возведения в степень, можно вычислить инверсию. Так, в конечном поле F_{19}

$$2/7 = 2 \times 7^{(19-2)} = 2 \times 7^{17} = 465261027974414 \% 19 = 3$$
$$7/5 = 7 \times 5^{(19-2)} = 7 \times 5^{17} = 5340576171875 \% 19 = 9$$

Такое вычисление является относительно затратным, поскольку возводимая степень растет очень быстро. И по этой причине операция деления оказывается самой затратной. Чтобы сократить затраты, можно воспользоваться доступной в языке Python функцией `pow()`, выполняющей возведение в степень. В языке Python вызов функции `pow(7, 17)` приводит к такому же результату, что и операция `7**17`. Но у функции `pow()` имеется необязательный третий аргумент, позволяющий повысить эффективность вычисления. Так, если указать третий аргумент, функция `pow()` выполнит операцию по модулю, используя значение этого аргумента. Следовательно, в результате вызова функции `pow(7,17,19)` будет получен такой же результат, как и при выполнении операции `7**17%19`, но сделано это будет быстрее, поскольку функция взятия модуля выполняется после каждого цикла умножения.

Упражнение 8

Решите следующие задачи в конечном поле F_{31}.

- 3 / 24
- 17^{-3}
- $4^{-4} \times 11$

Упражнение 9

Напишите соответствующий метод `__truediv__()`, реализующий операцию деления двух элементов конечного поля.

Следует, однако, иметь в виду, что в версии Python 3 операция деления разделена по двум функциям: `__truediv__()` и `__floordiv__()`. Первая функция выполняет обычную операцию деления, а вторая — операцию целочисленного деления.

Переопределение операции возведения в степень

И в завершение этой главы необходимо определить метод `__pow__()`, в котором должна быть предусмотрена правильная обработка отрицательных показателей степени. Например, a^{-3} должен быть элементом конечного поля, но в текущем коде такой случай не предусмотрен. Итак, требуется, чтобы выполнялись следующие действия:

```
>>> from ecc import FieldElement
>>> a = FieldElement(7, 13)
>>> b = FieldElement(8, 13)
>>> print(a**-3==b)
True
```

К сожалению, метод `__pow__()` определен таким образом, что отрицательные показатели степени в нем не обрабатываются должным образом, ведь в качестве второго параметра встроенной в Python функции `pow()` должно быть непременно указано положительное значение.

Правда, для этой задачи можно воспользоваться соответствующим математическим аппаратом. Из малой теоремы Ферма известно, что

$$a^{p-1} = 1$$

Это значит, что умножать на a^{p-1} можно столько раз, сколько угодно. Таким образом, над элементом a^{-3} можно выполнить следующую операцию:

$$a^{-3} = a^{-3} \times a^{p-1} = a^{p-4}$$

Именно таким образом можно обрабатывать отрицательные показатели степени. А наивная реализация метода __pow__() будет выглядеть следующим образом:

```
class FieldElement:
...
  def __pow__(self, exponent):
    n = exponent
    while n < 0:
      n += self.prime - 1     ❶
    num = pow(self.num, n, self.prime)     ❷
    return self.__class__(num, self.prime)
```

❶ Сложение выполняется до тех пор, пока показатель степени не станет положительным.

❷ Для повышения эффективности можно воспользоваться встроенной в Python функцией pow().

Правда, можно поступить еще лучше, ведь мы уже знаем, как превратить отрицательное число в положительное, используя знакомую нам операцию %! А кроме того, мы можем сократить очень большие показатели степени, принимая во внимание, что $a^{p-1} = 1$. Благодаря этому снижается нагрузка на функцию pow(), как показано ниже.

```
class FieldElement:
...
  def __pow__(self, exponent):
    n = exponent % (self.prime - 1)     ❶
    num = pow(self.num, n, self.prime)
    return self.__class__(num, self.prime)
```

❶ Показатель степени приводится к значению в пределах от 0 до $p-2$ включительно.

Заключение

В этой главе были описаны конечные поля и показано, как реализовать их на языке Python. В главе 3 нам еще предстоит воспользоваться конечными полями для криптографии по эллиптическим кривым. А в следующей главе мы рассмотрим математическую составляющую криптографии по эллиптическим кривым — собственно эллиптические кривые.

ГЛАВА 2

Эллиптические кривые

В этой главе речь пойдет об эллиптических кривых. А в главе 3 мы соединим их с конечными полями для криптографии по эллиптическим кривым.

Как и конечные поля, эллиптические кривые могут выглядеть пугающе, если вы не видели их прежде. Но на самом деле их математический аппарат также не очень сложен. Большая часть из того, что следует знать об эллиптических кривых, вам известна из алгебры. В этой главе рассматривается, что представляют собой подобные кривые и что с ними можно делать.

Определение эллиптических кривых

Эллиптические кривые (elliptic curve) описываются уравнениями, как и многие другие кривые, известные из курса основ алгебры. На двухмерной плоскости, где координаты x откладываются по горизонтальной оси, а координаты y — по вертикальной, они могут быть представлены в следующей форме:

$$y^2 = x^3 + ax + b$$

С подобными уравнениями вам, скорее всего, приходилось иметь дело. Например, из начального курса алгебры вам должно быть известно следующее линейное уравнение, описывающее прямую линию на координатной плоскости:

$$y = mx + b$$

Возможно, вы даже помните, что константа m в этом уравнении определяет *наклон* прямой линии, а константа b — ее *точку пересечения с осью y*. Линейное уравнение можно представить как график на координатной плоскости, рис. 2.1.

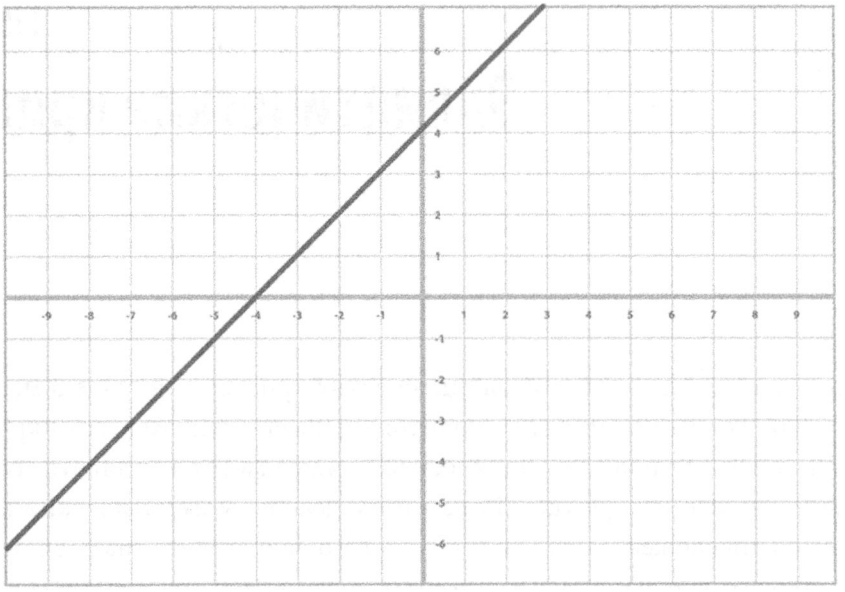

Рис. 2.1. Прямая, построенная по линейному уравнению

Аналогично вам должны быть знакомы квадратное уравнение и график его кривой (рис. 2.2):

$$y = ax^2 + bx + c$$

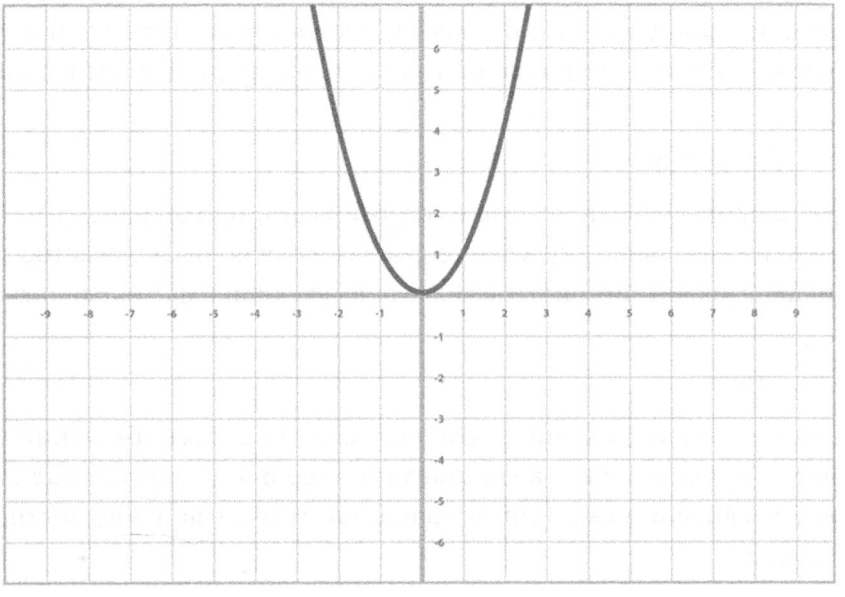

Рис. 2.2. Парабола, построенная по квадратному уравнению

А возможно, вам встречались алгебраические уравнения более высокого порядка, например кубическое уравнение и график его кривой (рис. 2.3):

$$y = ax^3 + bx^2 + cx + d$$

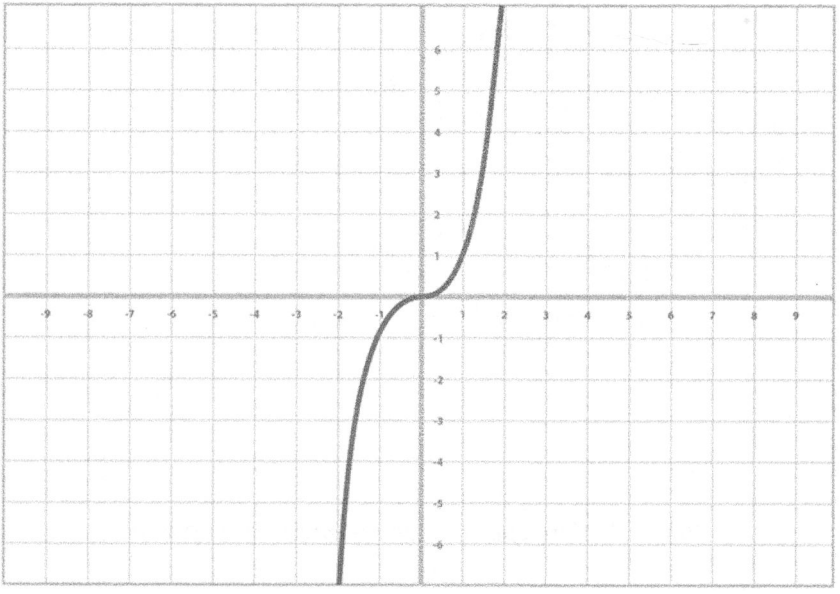

Рис. 2.3. Кубическая парабола, построенная по кубическому уравнению

Уравнение, описывающее эллиптическую кривую, в этом отношении ничем особенным не отличается, как показано ниже.

$$y^2 = x^3 + ax + b$$

Единственное отличие эллиптической кривой от кубической параболы, приведенной на рис. 2.3, заключается в члене y^2, находящемся в левой части ее уравнения. Благодаря этому график эллиптической кривой располагается симметрично относительно оси x, как показано на рис. 2.4.

Кроме того, эллиптическая кривая обладает меньшей крутизной, чем кубическая парабола. И этим она снова обязана члену y^2 в левой части ее уравнения. Иногда эллиптические кривые могут быть расчлененными, как показано на рис. 2.5.

Если угодно, эллиптическую кривую можно рассматривать как кубическую параболу (рис. 2.6), одна часть которой, находящаяся выше оси x, спрямлена (рис. 2.7), а другая часть — ниже оси x, зеркально отображена (рис. 2.8).

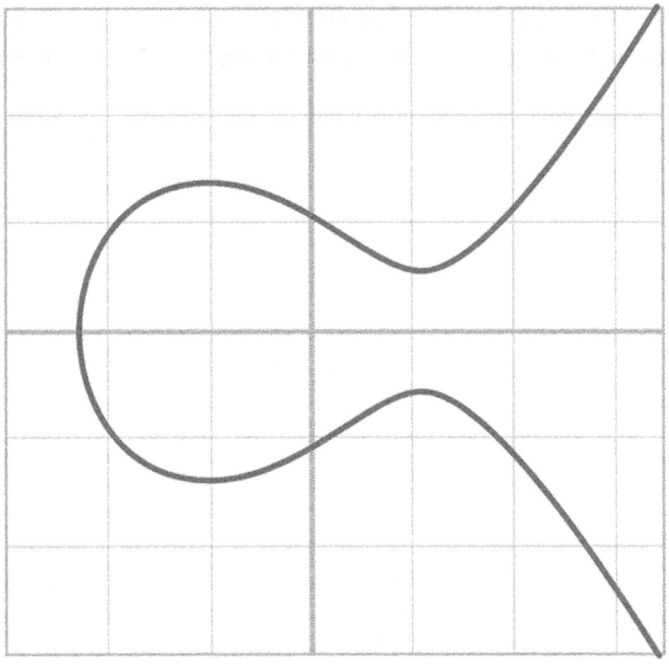

Рис. 2.4. Непрерывная эллиптическая кривая

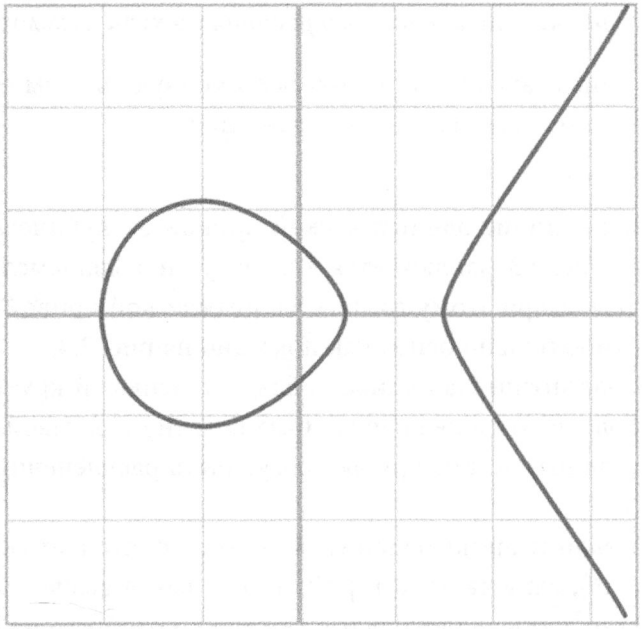

Рис. 2.5. Расчлененная эллиптическая кривая

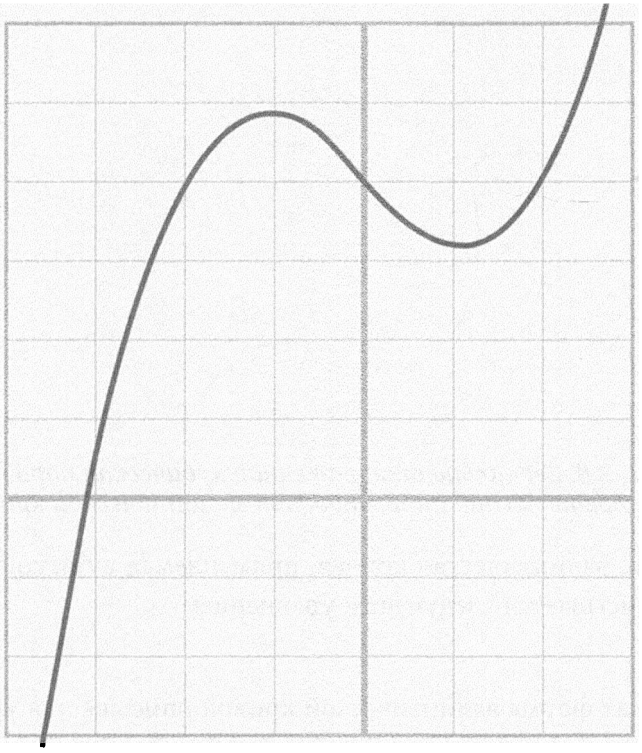

Рис. 2.6. Кубическая парабола на первой стадии построения эллиптической кривой

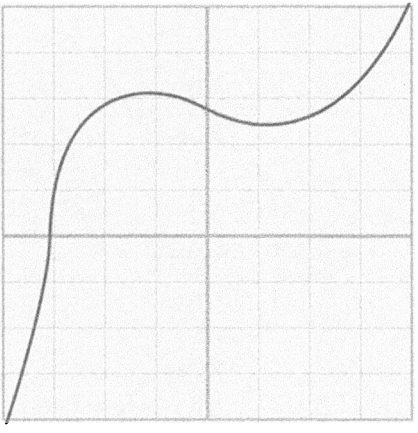

Рис. 2.7. Спрямленная кубическая парабола на второй стадии построения эллиптической кривой

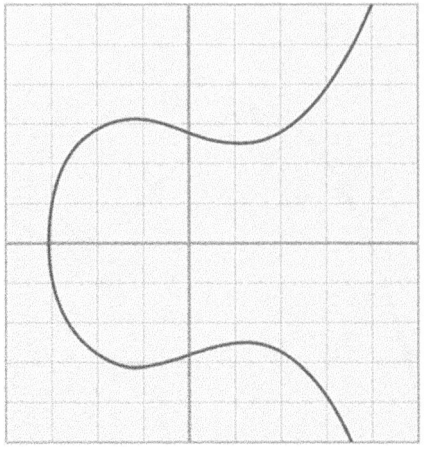

Рис. 2.8. Зеркально отображенная кубическая парабола на третьей стадии построения эллиптической кривой

В частности, эллиптическая кривая, применяемая в биткойне, называется *secp256k1* и описывается следующим уравнением:

$$y^2 = x^3 + 7$$

Каноническая форма эллиптической кривой описывается уравнением $y^2 = x^3 + ax + b$, поэтому упомянутая выше кривая определяется константами $a = 0$, $b = 7$ и выглядит так, как показано на рис. 2.9.

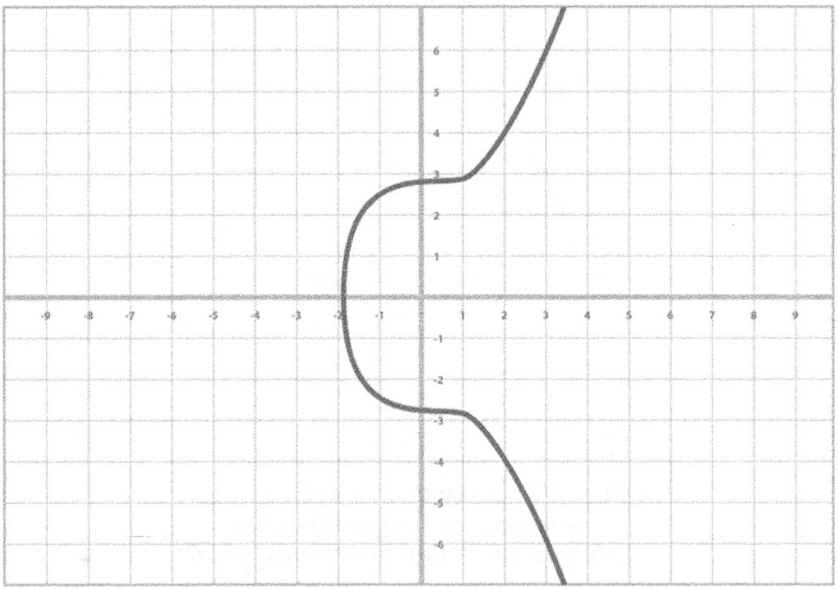

Рис. 2.9. Эллиптическая кривая secp256k1

Программная реализация эллиптических кривых на языке Python

По самым разным причинам, которые станут ясны в дальнейшем, нас больше интересуют не сами эллиптические кривые, а конкретные точки на них. Так, на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$, нас интересует точка с координатами (–1,1), поэтому определим класс `Point`, представляющий *точку* (point) на конкретной кривой. А поскольку эта кривая описывается уравнением $y^2 = x^3 + ax + b$, ее можно определить с помощью всего двух числовых констант, a и b, в одноименных полях данного класса, как показано ниже.

```
class Point:

  def __init__(self, x, y, a, b):
    self.a = a
    self.b = b
    self.x = x
    self.y = y
    if self.y**2 != self.x**3 + a * x + b: ❶
       raise ValueError('({}, {}) is not on the curve'.format(x, y))

    def __eq__(self, other):    ❷
      return self.x == other.x and self.y == other.y \
             and self.a == other.a and self.b == other.b
```

❶ Здесь проверяется, находится ли данная точка на кривой.

❷ Точки равны лишь в том и только в том случае, если они находятся на одной и той же кривой и имеют одинаковые координаты.

А теперь можно создать объекты типа `Point`, представляющие отдельные точки, как показано ниже. И если эти точки не находятся на одной и той же кривой, то происходит ошибка. Иными словами, в методе `__init__()` будет передано исключение, если проверяемая точка не окажется на кривой.

```
>>> from ecc import Point
>>> p1 = Point(-1, -1, 5, 7)
>>> p2 = Point(-1, -2, 5, 7)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "ecc.py", line 143, in __init__
```

```
raise ValueError('({}, {}) is not on the curve'.format(self.x, self.y))
ValueError: (-1, -2) is not on the curve[1]
```

Упражнение 1

Выясните, какие из приведенных ниже точек находятся на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$.

(2,4), (–1,–1), (18,77), (5,7)

Упражнение 2

Напишите для класса `Point` метод `__ne__()`, чтобы сравнить в нем два объекта данного класса на неравенство.

Сложение точек

Эллиптические кривые удобны тем, что позволяют выполнять *сложение точек* (point addition). Такая операция подразумевает получение третьей точки из двух исходных точек, находящихся на кривой. Она называется *сложением* (addition) потому, что во многом напоминает арифметическую операцию сложения. В частности, операция сложения точек коммутативна, т.е. сложение точек A и B дает такой же результат, как и сложение точек B и A.

Сложение точек определяется исходя из того, что каждая эллиптическая кривая пересекается прямой линией в одной (рис. 2.10) или же в трех точках (рис. 2.11), за исключением двух особых случаев.

В двух особых случаях прямая линия пересекает эллиптическую кривую вертикально (рис. 2.12) или по касательной к кривой (рис. 2.13). Мы еще вернемся в дальнейшем к этим двум случаям.

---

[1] Обратная трассировка стека (последний вызов указан последним):
Файл "<stdin>", строка 1, в *<имя_модуля>*
Файл "ecc.py", строка 143, в методе __init__()
возбуждено исключение типа ValueError('({}, {}) не находится на кривой'.format(self.x, self.y)) ValueError: (-1, -2) не находится на кривой

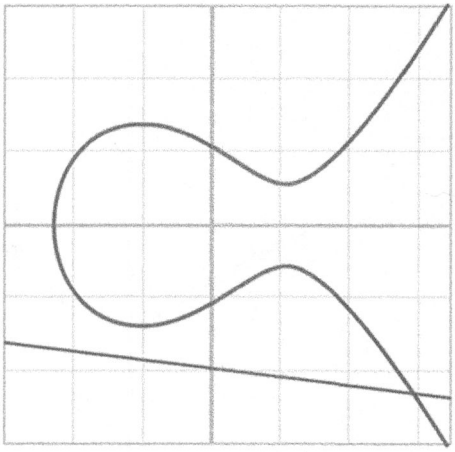

Рис. 2.10. Прямая линия пересекает эллиптическую кривую в одной точке

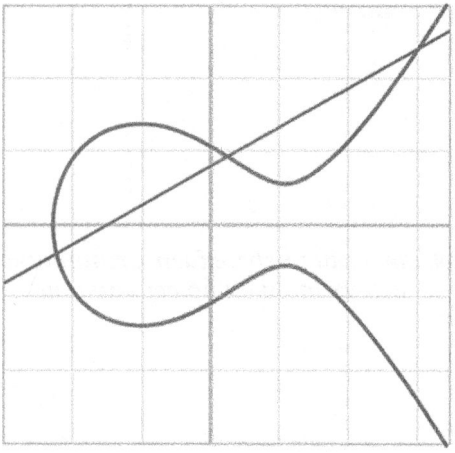

Рис. 2.11. Прямая линия пересекает эллиптическую кривую в трех точках

Таким образом, сложение точек можно определить на основании того факта, что прямые линии пересекают эллиптическую кривую в одной или трех точках. Прямая линия определяется двумя точками, и она должна пересекать эллиптическую кривую один или больше раз. Следовательно, третья точка пересечения, зеркально отображающаяся относительно оси x, является результатом сложения двух исходных точек.

Таким образом, чтобы получить сумму $P_1 + P_2$ любых двух точек $P_1 = (x_1,y_1)$ и $P_2 = (x_2,y_2)$, необходимо выполнить следующие действия.

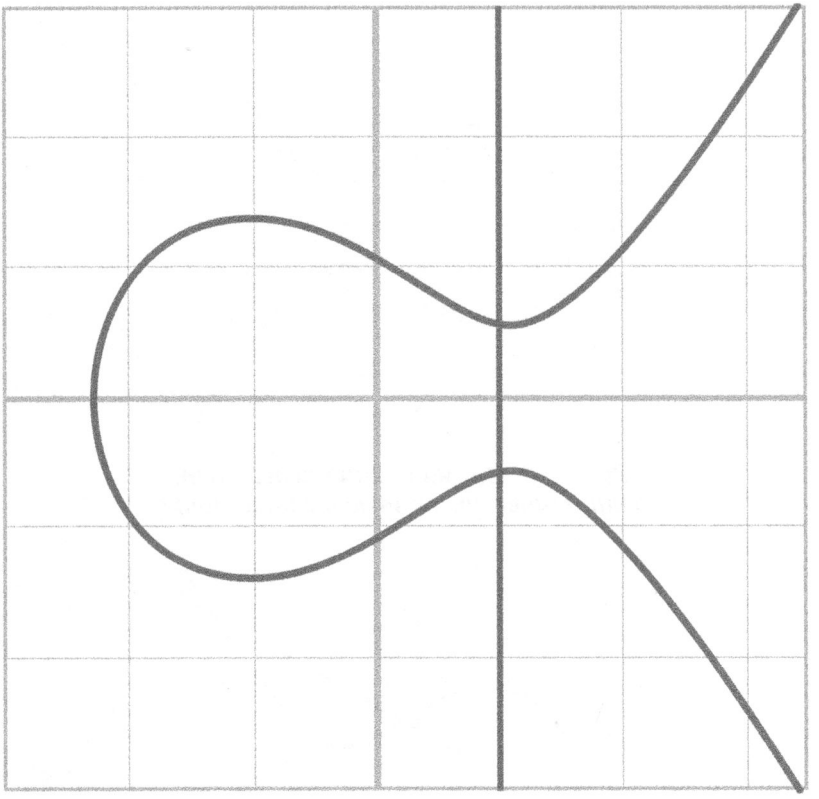

Рис. 2.12. Прямая линия пересекает эллиптическую кривую в двух точках по вертикали

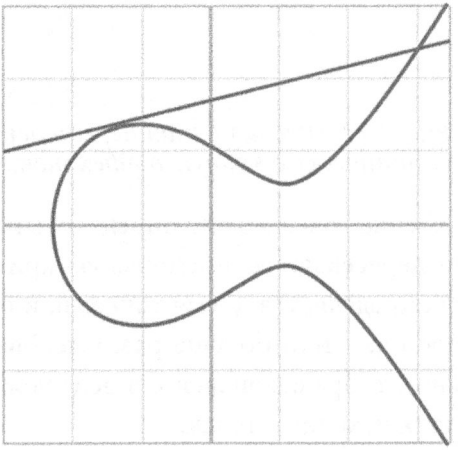

Рис. 2.13. Прямая линия пересекает эллиптическую кривую в двух точках по касательной к ней

- Найти точку, пересекающую эллиптическую кривую в третий раз, проведя прямую линию через исходные точки P_1 и P_2.
- Зеркально отобразить результирующую точку относительно оси x.

Описанные выше действия наглядно представлены на рис. 2.14.

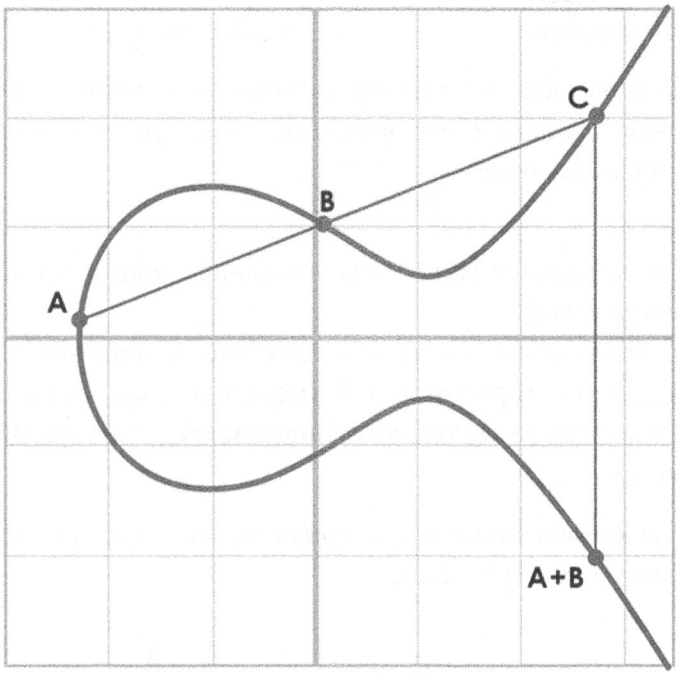

Рис. 2.14. Сложение точек

Сначала мы проводим прямую линию через суммируемые точки A и B. При этом прямая линия пересечет эллиптическую кривую в третьей точке C. Затем мы зеркально отображаем эту точку относительно оси x, получая в конечном итоге суммарную точку A + B, как показано на рис. 2.14.

Одно из свойств, которое мы собираемся воспользоваться, заключается в том, что сложение точек нелегко предсказать. Сложение точек нетрудно рассчитать по формуле, но вполне очевидно, что результат сложения двух точек может оказаться где угодно на кривой. Так, если вернуться к рис. 2.14, то сумма точек A + B оказывается справа от них, сумма точек A + C — где-то между точками A и C на оси x, а сумма точек B + C — слева от обеих точек. Поэтому в математике сложение точек называется *нелинейным*.

Математические основы сложения точек

Сложение точек удовлетворяет определенным свойствам, которые мы обычно связываем со сложением, в том числе следующим.

- Тождественность
- Коммутативность
- Ассоциативность
- Обратимость

Здесь *тождественность* (identity) означает наличие нуля, т.е. существует некоторая точка I, сложение которой с точкой A дает в итоге ту же самую точку A, как показано ниже.

$$I + A = A$$

Такая точка называется *бесконечно удаленной* (point at infinity) (причины этого поясняются далее).

С данным свойством тесно связано свойство *обратимости* (invertibility). В частности, для некоторой точки A имеется другая точка, $-A$, сложение с которой дает в итоге точку тождественности, как показано ниже.

$$A + (-A) = I$$

Наглядно на кривой такие точки можно представить как противоположные относительно оси x (рис. 2.15).

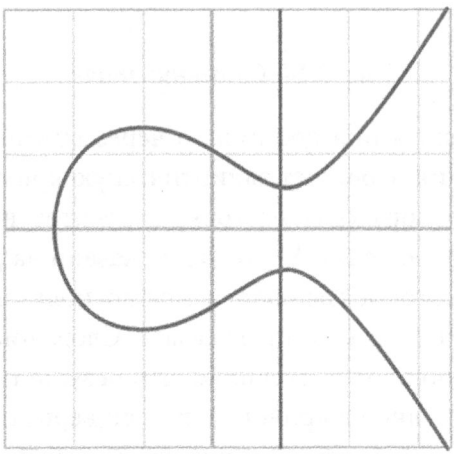

Рис. 2.15. Противоположные точки, образующиеся в результате пересечения прямой линией эллиптической кривой по вертикали

Точка *I* называется бесконечно удаленной именно потому, что на эллиптической кривой имеется еще одна точка, где прямая линия пересекается с этой кривой по вертикали в третий раз.

Коммутативность (commutativity) означает, что $A + B = B + A$. Это очевидно потому, что прямая линия, проведенная через точки A и B, пересекает эллиптическую кривую в третий раз в одном и том же месте независимо от того, в каком именно порядке это происходит.

Ассоциативность (associativity) означает, что $(A + B) + C = A + (B + C)$. Это свойство не совсем очевидно и служит причиной для переворота относительно оси *x*, как показано на рис. 2.16 и 2.17.

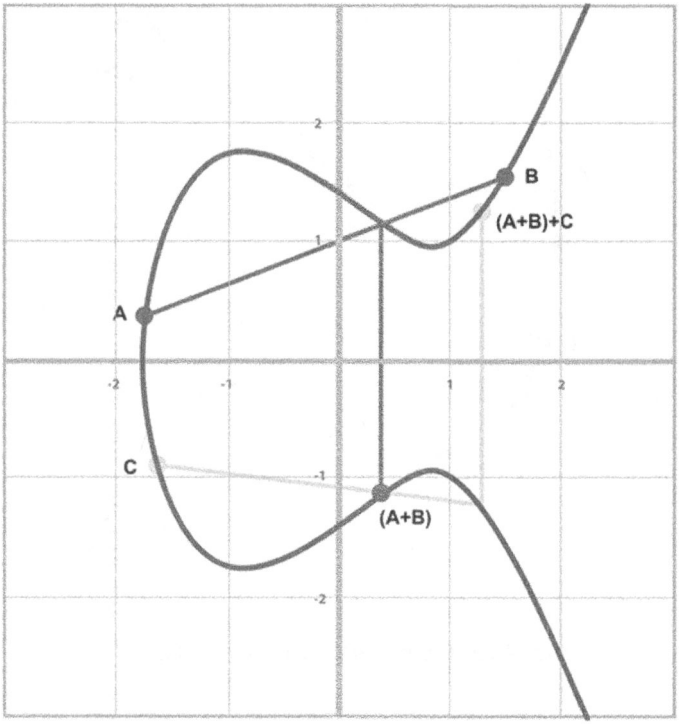

Рис. 2.16. Сложение точек (A + B) + C

Как показано на рис. 2.16 и 2.17, конечная точка оказывается той же самой. Иными словами, есть веские основания считать, что $(A + B) + C = A + (B + C)$. И хотя это не доказывает ассоциативность сложения точек, истинность данного утверждения показана наглядно.

Для программной реализации операции сложения точек, ее придется разделить на три стадии, где проверяются следующие условия.

1. Исходные точки находятся на проведенной вертикально прямой линии, т.е. используется точка тождественности.
2. Исходные точки находятся не на проведенной вертикально прямой линии, а в разных местах.
3. Исходные точки одинаковы.

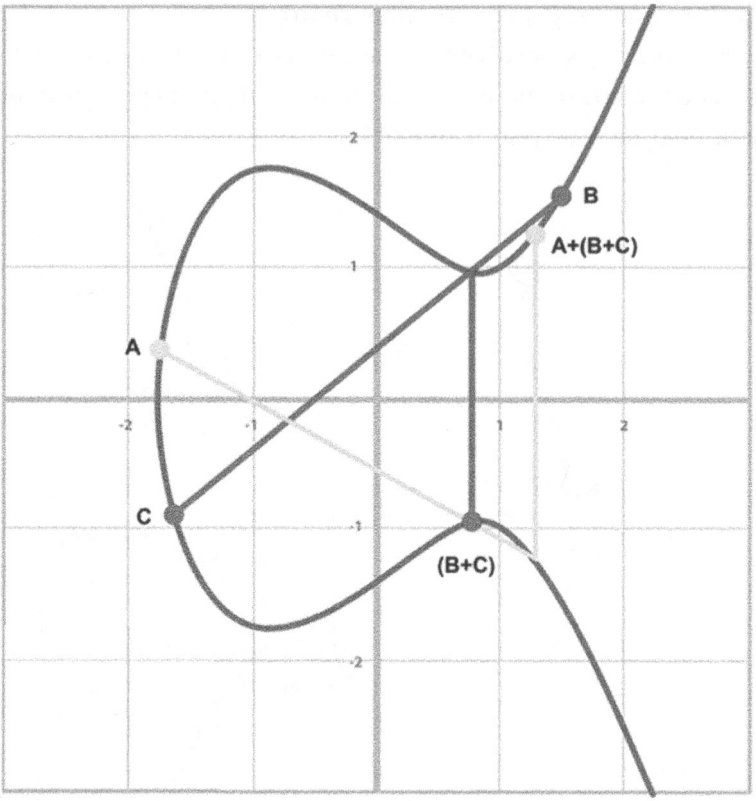

Рис. 2.17. Сложение точек A + (B + C)

Программная реализация операции сложения точек

Сначала необходимо обработать условие, связанное с наличием точки тождественности, т.е. бесконечно удаленной точки. Но поскольку реализовать бесконечность на языке Python не так-то просто, воспользуемся вместо этого значением `None`, обозначающим отсутствие конкретного значения. Итак, требуется, чтобы выполнялись следующие действия:

```
>>> from ecc import Point
>>> p1 = Point(-1, -1, 5, 7)
>>> p2 = Point(-1, 1, 5, 7)
>>> inf = Point(None, None, 5, 7)
>>> print(p1 + inf)
Point(-1,-1)_5_7
>>> print(inf + p2)
Point(-1,1)_5_7
>>> print(p1 + p2)
Point(infinity)
```

Чтобы добиться этого, необходимо сделать две вещи. Во-первых, следует внести незначительные коррективы в метод `__init__()`, поскольку в нем не проверяется, удовлетворяется ли уравнение, описывающее эллиптическую кривую, при наличии бесконечно удаленной точки. И во-вторых, необходимо перегрузить операцию сложения или метод `__add__()`, как это уже делалось в классе `FieldElement`. Ниже показано, как сделать и то, и другое.

```
class Point:

  def __init__(self, x, y, a, b):
    self.a = a
    self.b = b
    self.x = x
    self.y = y
    if self.x is None and self.y is None:     ❶
      return
    if self.y**2 != self.x**3 + a * x + b:
      raise ValueError('({}, {}) is not on the curve'.format(x, y))

  def __add__(self, other):     ❷
    if self.a != other.a or self.b != other.b:
      raise TypeError('Points {}, {} are not on the same curve'
          .format(self, other))
    if self.x is None:     ❸
      return other
    if other.x is None:     ❹
      return self
```

❶ Проверяемые здесь значения `None` координат *x* и *y* обозначают бесконечно удаленную точку. Обратите внимание на то, что следующий условный оператор `if` завершится неудачно, если не выполнить возврат из данного условного оператора.

❷ Здесь перегружается операция сложения (+).

❸ Значение `None` в поле `self.x` означает, что текущий объект `self` представляет бесконечно удаленную точку или аддитивное тождество. Поэтому в данном условном операторе возвращается другой объект `other`.

❹ Значение `None` в поле `other.x` означает, что другой объект `other` представляет бесконечно удаленную точку или аддитивное тождество. Поэтому в данном условном операторе возвращается текущий объект `self`.

Упражнение 3

Запрограммируйте проверку в том случае, если обе точки являются аддитивной инверсией, т.е. имеют одинаковые координаты x, но разные координаты y, а следовательно, через них проходит вертикальная линия. В результате такой проверки должна быть возвращена бесконечно удаленная точка.

Сложение точек, когда $X_1 \neq X_2$

Итак, рассмотрев случай, когда прямая линия пересекает эллиптическую кривую по вертикали, перейдем к обсуждению случая, когда исходные точки оказываются разными. Если исходные точки имеют разные координаты x, то и в этом случае их можно сложить по довольно простой формуле. Действуя интуитивно, найдем сначала наклон, образуемый обеими точками. Это можно сделать по формуле, известной из элементарной алгебры, как показано ниже.

$$P_1 = (x_1,y_1),\ P_2 = (x_2,y_2),\ P_3 = (x_3,y_3)$$
$$P_1 + P_2 = P_3$$
$$s = (y_2 - y_1)/(x_2 - x_1)$$

Здесь s обозначает *наклон*, с помощью которого можно рассчитать координату x_3. А зная координату x_3, можно рассчитать и координату y_3. Таким образом, положение точки P_3 получается по следующим формулам.

$$x_3 = s^2 - x_1 - x_2$$
$$y_3 = s(x_1 - x_3) - y_1$$

Напомним, что координата y_3 обозначает зеркальное отображение относительно оси x.

Вывод формулы для сложения точек

Допустим, что

$P_1 = (x_1,y_1)$, $P_2 = (x_2,y_2)$, $P_3 = (x_3,y_3)$

$P_1 + P_2 = P_3$

Требуется выяснить координаты точки P_3.

Итак, начнем с того, что прямая линия проходит через точки P_1 и P_2 и описывается следующими уравнениями:

$s = (y_2 - y_1)/(x_2 - x_1)$

$y = s(x - x_1) + y_1$

Второе уравнение описывает прямую линию, пересекающую обе точки, P_1 и P_2. Подставив это уравнение в уравнение, описывающее эллиптическую кривую, получим следующий результат:

$y^2 = x^3 + ax + b$

$y^2 = (s(x - x_1) + y_1)^2 = x^3 + ax + b$

Сведя все подобные члены данного уравнения, получим следующее полиномиальное уравнение.

$x^3 - s^2x^2 + (a + 2s^2x_1 - 2sy_1)x + b - s^2x_1^2 + 2sx_1y_1 - y_1^2 = 0$

Известно также, что решение данного уравнения дает координаты x_1, x_2 и x_3, а следовательно

$(x - x_1)(x - x_2)(x - x_3) = 0$

$x^3 - (x_1 + x_2 + x_3)x^2 + (x_1x_2 + x_1x_3 + x_2x_3)x - x_1x_2x_3 = 0$

Из предыдущего известно, что

$x^3 - s^2x^2 + (a + 2s^2x_1 - 2sy\sim1\sim)x + b - s^2x_1^2 + 2sx\sim1\sim y_1 - y_1^2 = 0$

На основании формул Виеты (`https://en.wikipedia.org/wiki/Vieta's_formulas`) можно установить, что коэффициенты многочлена оказываются одинаковыми, если одинаковы его корни. Первым интересующим нас коэффициентом является тот, который стоит перед членом x^2:

$-s^2 = -(x_1 + x_2 + x_3)$

Используя это уравнение, можно вывести формулу для определения координаты x_3 следующим образом:

$x_3 = s^2 - x_1 - x_2$

Эту формулу можно подставить в приведенную выше формулу для описания прямой линии:

$y = s(x - x_1) + y_1$

Но поскольку третью точку следует зеркально отобразить относительно оси x, правую часть данного уравнения следует инвертировать:

$y_3 = -(s(x_3 - x_1) + y_1) = s(x_1 - x_3) - y_1$

Что и требовалось доказать.

Упражнение 4

Где на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$, находится точка, получаемая в результате сложения исходных точек с координатами (2,5) + (–1,–1)?

Программная реализация операции сложения точек, когда $x_1 \neq x_2$

А теперь реализуем приведенные выше математические выкладки в своей библиотеке. По существу, это означает необходимость внести соответствующие коррективы в метод `__add__()` для проверки условия в том случае, когда $x_1 \neq x_2$. Для этой цели у нас имеются следующие формулы:

$$s = (y_2 - y_1)/(x_2 - x_1)$$

$$x_3 = s^2 - x_1 - x_2$$

$$y_3 = s(x_1 - x_3) - y_1$$

В конце данного метода должен быть возвращен экземпляр класса `Point`. Для упрощения подклассификации это делается с помощью метода `self.__class__()`.

Упражнение 5

Напишите метод `__add__()` для проверки условия в том случае, когда $x_1 \neq x_2$.

Сложение точек, когда $P_1 = P_2$

Если координаты x исходных точек одинаковы, а их координаты y различаются, эти точки располагаются одна напротив другой относительно оси x.

Как установлено ранее и практически усвоено в упражнении 3, это означает следующее:

$$P_1 = -P_2 \text{ или } P_1 + P_2 = \text{I}$$

А что произойдет, если $P_1 = P_2$? В этом случае придется сначала рассчитать прямую линию, *касательную* к эллиптической кривой в точке P_1, а затем найти точку, в которой эта линия пересекает данную кривую. Подобный случай рассматривался ранее и наглядно представлен на рис. 2.18.

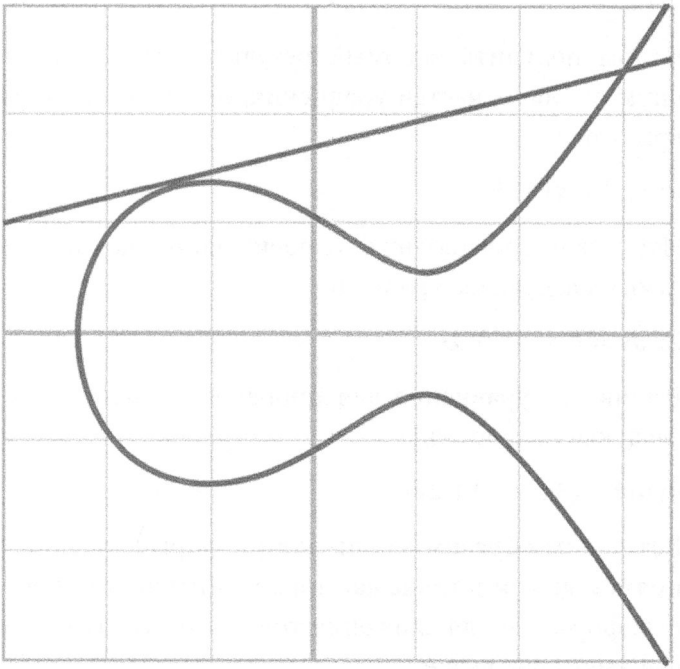

Рис. 2.18. Прямая линия, проведенная касательно к эллиптической кривой

И в этом случае необходимо определить наклон прямой линии в точке ее касания, как показано ниже.

$$P_1 = (x_1,y_1),\ P_3 = (x_3,y_3)$$
$$P_1 + P_1 = P_3$$
$$s = (3x_1^2 + a)/(2y_1)$$

Остальные формулы такие же, как и прежде, за исключением того, что $x_1 = x_2$, поэтому их можно объединить следующим образом:

$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$

Вывод формулы для расчета наклона прямой линии в точке касания эллиптической кривой

Формулу для расчета наклона прямой линии в точке касания эллиптической кривой можно вывести, используя более сложный математический аппарат дифференциального исчисления. Как известно, наклон прямой линии в заданной точке описывается следующим образом:

dy/dx

Чтобы получить искомый результат, придется взять производную от обеих частей уравнения, описывающего эллиптическую кривую:

$y^2 = x^3 + ax + b$

Итак, взяв производную от обеих частей данного уравнения, получим следующее уравнение:

$2y\ dy = (3x^2 + a)\ dx$

Решив это уравнение для производной dy/dx, в итоге получим следующую формулу:

$dy/dx = (3x^2 + a)/(\ 2y)$

Вот каким образом мы пришли к формуле для расчета наклона прямой линии в точке касания эллиптической кривой. А остальные формулы для сложения точек в рассматриваемом здесь случае выведены выше.

Упражнение 6

Где на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$, находится точка, получаемая сложением исходных точек с координатами (–1,–1) + (–1,–1)?

Программная реализация операции сложения точек, когда $P_1 = P_2$

Внесем для данного конкретного случая соответствующие коррективы в метод `__add__()`. Для этого у нас имеются приведенные ниже формулы, которые остается лишь реализовать непосредственно в коде.

$$s = (3x^2 + a)/(2y)$$
$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$

Упражнение 7

Напишите метод `__add__()` для проверки условия в том случае, когда $P_1 = P_2$.

Программная реализация операции сложения точек в еще одном исключительном случае

Имеется еще один исключительный случай, когда касательная к эллиптической кривой линия проведена по вертикали (рис. 2.19).

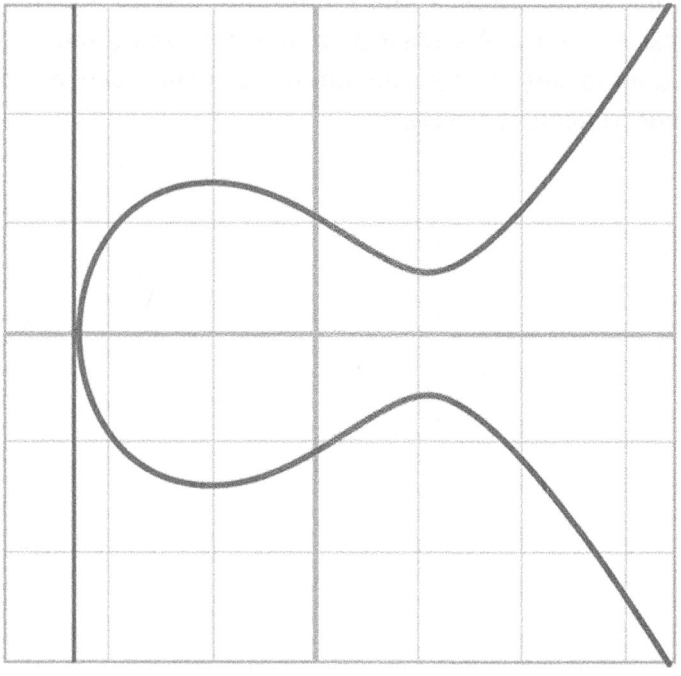

Рис. 2.19. Прямая линия, проведенная касательно к эллиптической кривой по вертикали

Это может произойти, если $P_1 = P_2$ и координаты y исходных точек равны нулю. И в этом случае расчет наклона прямой линии дает бесконечный

результат из-за наличия нуля в знаменателе соответствующей формулы. Этот особый случай программируется следующим образом:

```
class Point:
  ...
  def __add__(self, other):
    ...
    if self == other and self.y == 0 * self.x:     ❶
      return self.__class__(None, None, self.a, self.b)
```

❶ Если две исходные точки одинаковы, а их координаты *y* равны нулю, то возвращается бесконечно удаленная точка.

Заключение

В этой главе были описаны эллиптические кривые, рассмотрено их назначение и порядок построения, а также показано, каким образом реализуется операция сложения точек. А в главе 3 все понятия, представленные в главах 1 и 2, будут использованы совместно для разъяснения особенностей криптографии по эллиптическим кривым.

ГЛАВА 3

Криптография по эллиптическим кривым

В двух предыдущих главах был описан основной математический аппарат рассматриваемого здесь предмета. В них рассматривались понятия конечных полей и эллиптических кривых. А в этой главе оба эти понятия будут использованы совместно для описания криптографии по эллиптическим кривым. В частности, нам предстоит построить примитивы, необходимые для подписания и верификации сообщений, т.е. двух основных операций, выполняемых в биткойне.

Эллиптические кривые над вещественными числами

В главе 2 было показано, как эллиптическая кривая выглядит на координатной плоскости, поскольку она была построена над *вещественными* числами. И это не просто целые или даже рациональные, а именно вещественные числа. В частности, число π, квадратный корень из 2, e + корень 7-й степени из 19 — все это примеры вещественных чисел.

Это оказалось возможным потому, что вещественные числа также являются полями. Но, в отличие от *конечных* полей, количество вещественных чисел *бесконечно*. А в остальном они обладают теми же свойствами, перечисленными ниже.

1. Если a и b принадлежат множеству, то ему принадлежат и операции $a + b$ и $a \times b$.
2. Нуль существует и обладает свойством $a + 0 = a$.
3. Единица существует и обладает свойством $a \times 1 = a$.

4. Если a принадлежит множеству, то ему принадлежат и $-a$, определяемое как значение, удовлетворяющее выражению $a + (-a) = 0$.
5. Если a не равно 0 и принадлежит множеству, то ему принадлежат и a^{-1}, определяемое как значение, удовлетворяющее выражению $a \times a^{-1} = 1$.

Очевидно, что все эти свойства истинны. В частности, обычные операции сложения и умножения вполне применимы, нулевой (0) и единичный (1) элементы аддитивного и мультипликативного тождеств существуют, $-x$ является аддитивной инверсией, а $1/x$ — мультипликативной инверсией.

Построить график кривой над вещественными числами нетрудно. Например, по уравнению $y^2 = x^3 + 7$ можно построить график кривой, приведенный на рис. 3.1.

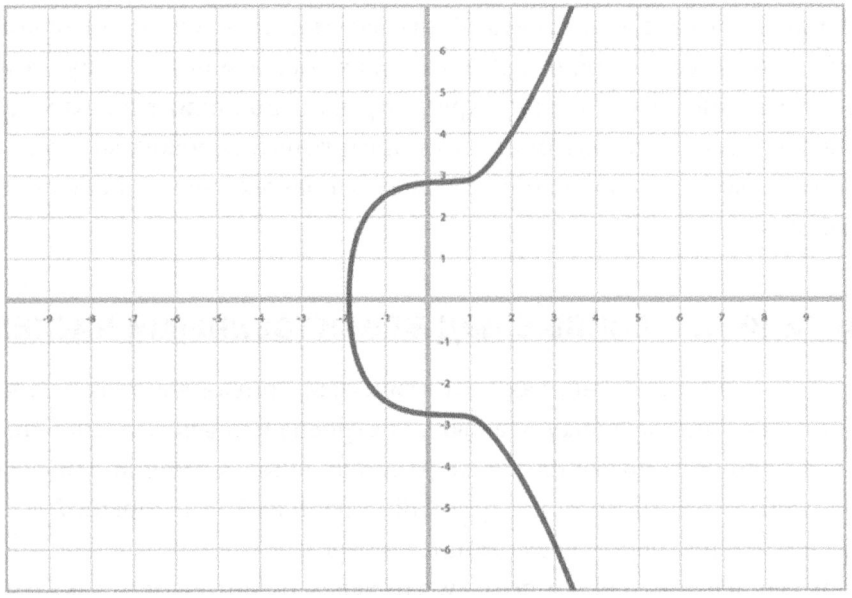

Рис. 3.1. График эллиптической кривой secp256k1, построенный над вещественными числами

Оказывается, что уравнения для сложения точек можно использовать над любым полем, включая конечные поля, описанные в главе 1. Единственное отличие состоит в том, что для этого придется применить операции сложения, вычитания, умножения и деления в конечном поле, определенные в главе 1, а не их обычные арифметические аналоги, выполняемые над вещественными числами.

Эллиптические кривые над конечными полями

Так как же выглядит эллиптическая кривая над конечным полем? Рассмотрим в качестве примера уравнение $y^2 = x^3 + 7$ над F_{103}. Вычислив обе его части, можно определить, находится ли точка с координатами (17,64) на кривой, описываемой данным уравнением, как показано ниже.

$y^2 = 64^2 \% 103 = 79$

$x^3 + 7 = (17^3+7) \% 103 = 79$

Таким образом, используя математический аппарат конечных полей, мы убедились, что данная точка действительно находится на кривой, описываемой приведенным выше уравнением. А поскольку это уравнение вычисляется над конечным полем, построенный по нему график выглядит совсем иначе (рис. 3.2).

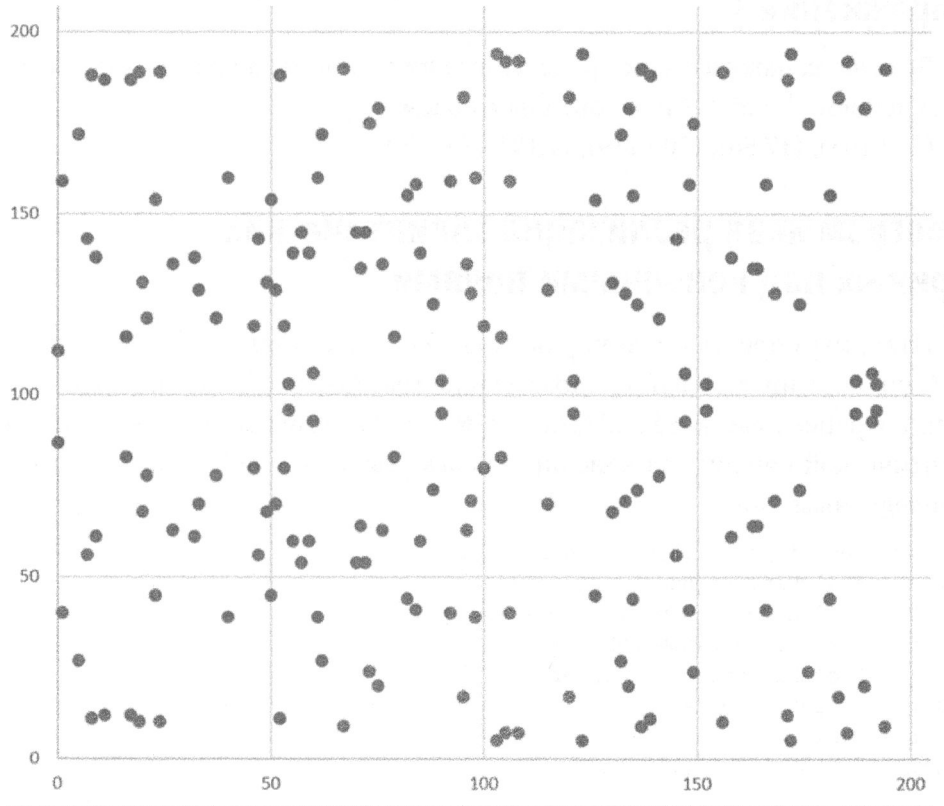

Рис. 3.2. График эллиптической кривой, построенной над конечным полем

Как видите, этот график больше похож на беспорядочное скопление точек, как от выстрела из дробовика, чем на плавную кривую. Единственная закономерность в нем — симметричность относительно середины, образующаяся благодаря члену y^2 в уравнении, описывающем данную кривую. Этот график симметричен не относительно оси x, как график кривой над вещественными числами, а относительно некого положительного значения на оси y, поскольку в конечном поле отсутствуют отрицательные числа.

Примечательно, однако, что в тех же самых уравнениях для сложения точек можно применять операции сложения, вычитания, умножения, деления и возведения в степень, определенные для конечных полей, и они будут действовать нормально. И хотя на первый взгляд это может показаться неожиданным, подобные закономерности проявляются в абстрактной математике, несмотря на отличия от знакомых вам традиционных методов расчета.

Упражнение 1

Выясните, находятся ли приведенные ниже точки на кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

(192,105), (17,56), (200,119), (1,193), (42,99)

Программная реализация эллиптических кривых над конечными полями

Итак, мы определили точку на эллиптической кривой и операции +, – ,* и / для конечных полей. Следовательно, мы можем объединить оба определенных ранее класса, `FieldElement` и `Point`, чтобы создавать точки на эллиптической кривой над конечным полем, выполняя действия, аналогичные приведенным ниже.

```
>>> from ecc import FieldElement, Point
>>> a = FieldElement(num=0, prime=223)
>>> b = FieldElement(num=7, prime=223)
>>> x = FieldElement(num=192, prime=223)
>>> y = FieldElement(num=105, prime=223)
>>> p1 = Point(x, y, a, b)
>>> print(p1)
Point(192,105)_0_7 FieldElement(223)
```

Для инициализации объекта класса `Point` выполняется следующий фрагмент кода из данного класса:

```
class Point:

  def __init__(self, x, y, a, b):
    self.a = a
    self.b = b
    self.x = x
    self.y = y
    if self.x is None and self.y is None:
      return
    if self.y**2 != self.x**3 + a * x + b:
      raise ValueError('({}, {}) is not on the curve'.format(x, y))
```

Но вместо целочисленных операций сложения (+), умножения (*), возведения в степень (**) и сравнения на неравенство (!=) здесь применяются равнозначные им методы `__add__()`, `__mul__()`, `__pow__()` и `__ne__()` из класса `FiniteField` соответственно. Построив библиотеку для криптографии по эллиптическим кривым, мы можем вычислять одно и то же уравнение с разными определениями основных арифметических операций.

Мы уже реализовали два класса, требующихся для реализации точек эллиптических кривых над конечным полем. Но для того, чтобы проверить проделанную нами работу, было бы полезно создать тестовый набор. Это можно сделать, используя результаты, полученные при выполнении упражнения 1, как показано ниже.

```
class ECCTest(TestCase):

  def test_on_curve(self):
    prime = 223
    a = FieldElement(0, prime)
    b = FieldElement(7, prime)
    valid_points = ((192, 105), (17, 56), (1, 193))
    invalid_points = ((200, 119), (42, 99))
    for x_raw, y_raw in valid_points:
      x = FieldElement(x_raw, prime)
      y = FieldElement(y_raw, prime)
      Point(x, y, a, b)     ❶
    for x_raw, y_raw in invalid_points:
      x = FieldElement(x_raw, prime)
      y = FieldElement(y_raw, prime)
      with self.assertRaises(ValueError):
          Point(x, y, a, b)     ❶
```

❶ Здесь объекты класса `FieldElement` передаются конструктору класса `Point` для инициализации, где, в свою очередь, применяются все арифметические операции, перегружаемые в классе `FieldElement`.

Этот тест можно выполнить теперь следующим образом:

```
>>> import ecc
>>> from helper import run    ❶
>>> run(ecc.ECCTest('test_on_curve'))
.
----------------------------------------------------------------------
Ran 1 test in 0.001s[1]
OK
```

❶ В состав модуля `helper` входит ряд очень удобных служебных функций, включая функцию `run()`, позволяющую выполнять модульные тесты по отдельности.

Сложение точек над конечными полями

Для вычислений над полями можно пользоваться теми же уравнениями, в том числе следующим линейным уравнением:

$$y = mx + b$$

Оказывается, что "линия" над конечным полем выглядит совсем не так, как можно было бы ожидать (рис. 3.3). Тем не менее приведенное выше линейное уравнение действует исправно, и поэтому мы можем вычислить координату *y*, если задана координата *x*.

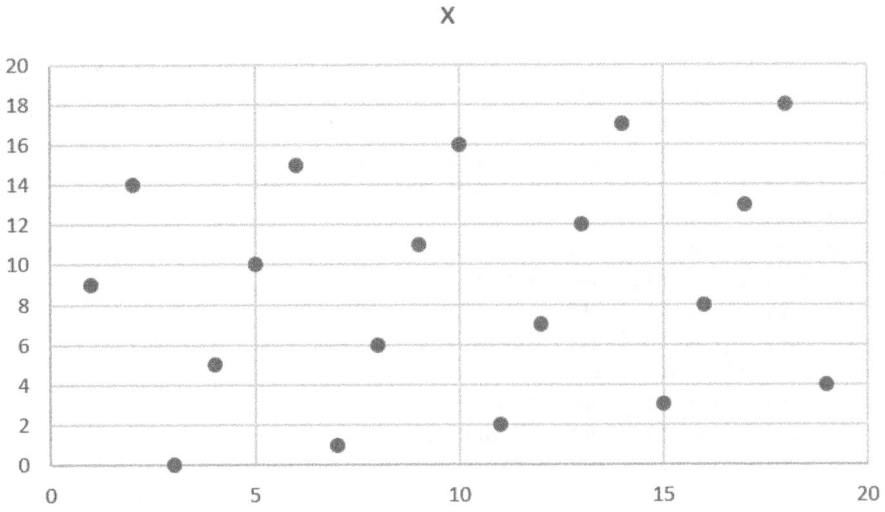

Рис. 3.3. График прямой линии, построенный над конечным полем

[1] Выполнен 1 тест за 0,001 с

Примечательно, что операция сложения точек действует и над конечными полями, поскольку над всеми конечными полями действует сложение точек эллиптической кривой! Тем же самые формулы, которыми мы пользовались раньше для расчета сложения точек над вещественными числами, оказываются пригодными и над конечными полями. Так, ниже приведены формулы для сложения точек, когда $x_1 \neq x_2$.

$$P_1 = (x_1,y_1), P_2 = (x_2,y_2), P_3 = (x_3,y_3)$$
$$P_1 + P_2 = P_3$$
$$s = (y_2 - y_1)/(x_2 - x_1)$$
$$x_3 = s^2 - x_1 - x_2$$
$$y_3 = s(x_1 - x_3) - y_1$$

А для сложения точек, когда $P_1 = P_2$, подходят следующие формулы:

$$P_1 = (x_1,y_1), P_3 = (x_3,y_3)$$
$$P_1 + P_1 = P_3$$
$$s = (3x_1^2 + a)/(2y_1)$$
$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$

Все приведенные выше уравнения пригодны для эллиптических кривых над конечными полями. Тем самым закладывается основание для создания некоторых криптографических примитивов.

Программная реализация операции сложения точек над конечными полями

Мы создали класс `FieldElement` так, чтобы определить в нем методы `__add__()`, `__sub__()`, `__mul__()`, `__truediv__()`, `__pow__()`, `__eq__()` и `__ne__()`. Поэтому достаточно инициализировать объект класса `Point` объектами типа `FieldElement`, чтобы успешно выполнить операцию сложения точек, как показано ниже.

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(num=0, prime=prime)
>>> b = FieldElement(num=7, prime=prime)
>>> x1 = FieldElement(num=192, prime=prime)
>>> y1 = FieldElement(num=105, prime=prime)
>>> x2 = FieldElement(num=17, prime=prime)
```

```
>>> y2 = FieldElement(num=56, prime=prime)
>>> p1 = Point(x1, y1, a, b)
>>> p2 = Point(x2, y2, a, b)
>>> print(p1+p2)
Point(170,142)_0_7 FieldElement(223)
```

Упражнение 2

Выполните сложение приведенных ниже точек для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

- (170,142) + (60,139)
- (47,71) + (17,56)
- (143,98) + (76,66)

Упражнение 3

Выполните расширение класса `ECCTest` для проверки правильности операций сложения точек из предыдущего упражнения. Присвойте методу такой проверки имя `test_add`.

Скалярное умножение для эллиптических кривых

Благодаря возможности складывать точку с самой собой можно ввести следующую новую форму записи:

$$(170,142) + (170,142) = 2 \times (170,142)$$

Аналогично точку можно прибавить снова благодаря свойству ассоциативности:

$$2 \times (170,142) + (170,142) = 3 \times (170, 142)$$

Это можно делать столько раз, сколько потребуется. Такая операция называется *скалярным умножением* (scalar multiplication) и отличается наличием *скалярного* числа перед координатами точки. Скалярное умножение оказывается возможным потому, что мы определили операцию многократного сложения точек как ассоциативную.

Скалярное умножение отличается, в частности, тем, что оно непредсказуемо без вычисления (рис. 3.4).

Каждая точка на рис. 3.4 помечена числом, обозначающим, сколько раз она была сложена. Как видите, в итоге получается график, похожий на

совершенно беспорядочное скопление точек, как от выстрела из дробовика. А объясняется это тем, что операция сложения точек носит нелинейный характер и не так-то просто вычисляется. Выполнить скалярное умножение нетрудно, чего нельзя сказать о противоположной ему операции деления. Это так называемая *задача дискретного логарифмирования*, положенная в основу криптографии по эллиптическим кривым.

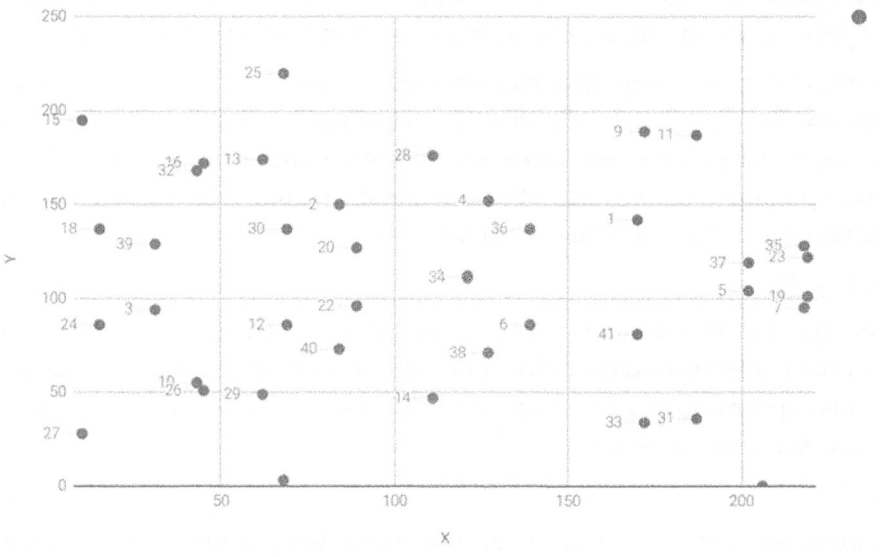

Рис. 3.4. Результаты скалярного умножения точки (170,142) для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}

Еще одно отличительное свойство скалярного умножения состоит в том, что на определенной стадии умножения мы доходим до бесконечно удаленной точки (напомним, что бесконечно удаленная точка — это нулевой элемент аддитивного тождества или просто 0). Так, если имеется точка G, которая скалярно умножается до тех пор, пока не будет достигнута бесконечно удаленная точка, то в конечном итоге получается следующее множество точек:

{ G, 2G, 3G, 4G, ... nG }, где nG = 0

Такое множество называется *группой*, а поскольку n конечно, то получается *конечная группа*, а точнее — *конечная циклическая группа*. С математической точки зрения группы любопытны тем, что они отличаются регулярным поведением по отношению к операции сложения, как показано ниже.

G + 4G = 5G или aG + bG = (a + b)G

Сочетая с математическими свойствами групп то обстоятельство, что выполнить скалярное умножение легко в одном направлении, тогда как в другом — трудно, мы в итоге получаем именно то, что и требуется для криптографии по эллиптическим кривым.

Происхождение названия задачи дискретного логарифмирования

У вас может возникнуть резонный вопрос: почему задача обращения скалярного *умножения* называется задачей дискретного *логарифмирования*?

Мы назвали ранее операцию над точками "сложением", но с тем же успехом ее можно было бы назвать "точечной операцией". Как правило, новая операция, определяемая в математике, обозначается оператором-точкой (×). Этот оператор применяется и для обозначения операции умножения, что иногда помогает мыслить категориями умножения:

$P_1 \times P_1 = P_3$

Многократное умножение сродни возведению в степень. Когда же выполняется скалярное умножение, названное ранее многократным "сложением точек", оно превращается в скалярное возведение в степень, если рассматривать его как "умножение точек":

$P^7 = Q$

В данном контексте задача дискретного логарифмирования означает возможность обратить приведенное выше уравнение, получив в итоге следующее логарифмическое уравнение:

$\log_P Q = 7$

У логарифма в левой части этого уравнения отсутствует аналитически вычислимый алгоритм. Это означает, что неизвестна формула, которую можно было бы подставить, чтобы получить общее решение данной задачи. И хотя это кажется не совсем ясным, справедливости ради все же следует сказать, что данную задачу можно было бы назвать задачей "дискретного деления точек", а не дискретного логарифмирования.

Упражнение 4

Выполните скалярное умножение приведенных ниже точек для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

- $2 \times (192,105)$
- $2 \times (143,98)$

- $2 \times (47,71)$
- $4 \times (47,71)$
- $8 \times (47,71)$
- $21 \times (47,71)$

Еще раз о скалярном умножении

Скалярное умножение означает сложение одной и той же точки с самой собой некоторое количество раз. Главным основанием для применения скалярного умножения в криптографии с открытым ключом служит то обстоятельство, что скалярное умножение на эллиптических кривых очень трудно обратить. Так, в предыдущем упражнении вам пришлось вычислять скалярное произведение точек $s \times (47,71)$ в конечном поле F_{223} для s в пределах от 1 до 21. Ниже приведены результаты такого вычисления.

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x = FieldElement(47, prime)
>>> y = FieldElement(71, prime)
>>> p = Point(x, y, a, b)
>>> for s in range(1,21):
... result = s*p
... print('{}*(47,71)=({},{})'.format(s,result.x.num,result.y.num))
1*(47,71)=(47,71)
2*(47,71)=(36,111)
3*(47,71)=(15,137)
4*(47,71)=(194,51)
5*(47,71)=(126,96)
6*(47,71)=(139,137)
7*(47,71)=(92,47)
8*(47,71)=(116,55)
9*(47,71)=(69,86)
10*(47,71)=(154,150)
11*(47,71)=(154,73)
12*(47,71)=(69,137)
13*(47,71)=(116,168)
14*(47,71)=(92,176)
15*(47,71)=(139,86)
16*(47,71)=(126,127)
17*(47,71)=(194,172)
```

```
18*(47,71)=(15,86)
19*(47,71)=(36,112)
20*(47,71)=(47,152)
```

Если внимательно рассмотреть приведенные выше числа, то в них не удается обнаружить какой-нибудь по-настоящему заметной закономерности, присущей скалярному умножению. В частности, координаты x и y умножаемых точек не всегда увеличиваются или уменьшаются. Единственная закономерность прослеживается в множителях 10 и 11, где координаты x равны, а также в множителях 8, 9, 12, 13 и т.д. И это объясняется тем, что $21 \times (47,71) = 0$.

Скалярное умножение выглядит довольно произвольным, что придает данному уравнению *асимметричность*. Асимметричную задачу легко решить в одном направлении, но трудно — в обратном. Например, вычислить скалярное произведение $12 \times (47,71)$ достаточно просто. Но если представить его как

$$s \times (47,71) = (194,172)$$

то можно ли найти решение для s? Мы можем найти приведенные выше результаты, но лишь благодаря тому, что у нас имеется небольшая группа. Как будет показано далее, в разделе "Определение кривой для биткойна", для намного бо́льших чисел задача дискретного логарифмирования становится трудноразрешимой.

Математические группы

К этому вопросу нас подводят рассмотренные ранее математические понятия (конечных полей, эллиптических кривых и их сочетания). Для криптографии с открытым ключом нам требуется сформировать конечные циклические группы. Оказывается, что если взять генераторную точку из эллиптической кривой над конечным полем, то с ее помощью можно сформировать конечную циклическую группу.

В отличие от полей, у групп имеется лишь одна операция. В данном случае это операция сложения точек. У групп имеется также ряд свойств: замкнутость, обратимость, коммутативность, ассоциативность и, наконец, требуемая тождественность. Рассмотрим эти свойства по отдельности, начиная с последнего.

Тождественность

Нетрудно догадаться, что тождественность определяется как бесконечно удаленная точка, которая гарантированно находится в группе, поскольку мы формируем группу, доходя до бесконечно удаленной точки. Таким образом:

$$0 + A = A$$

Здесь 0 называется бесконечно удаленной точкой потому, что она помогает наглядно представить решаемую математическую задачу (рис. 3.5).

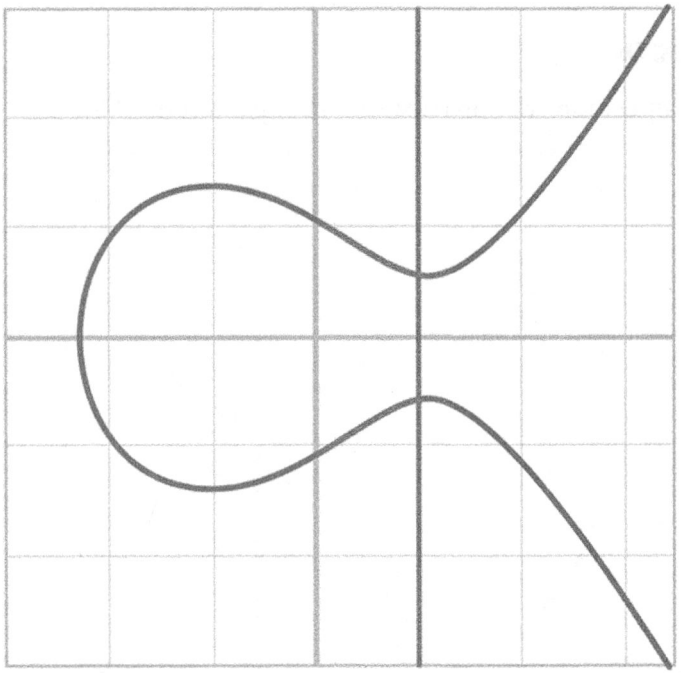

Рис. 3.5. Вертикальная прямая "пересекает" в третий раз кривую в бесконечно удаленной точке

Замкнутость

Это свойство, вероятно, проще всего обосновать, поскольку мы сформировали группу, прежде всего, многократно прибавляя точку G. Так, если в группе имеются два разных элемента, например

$$aG + bG,$$

то результат их сложения, как известно, окажется следующим:

$$(a + b)G$$

А откуда известно, что этот элемент находится в группе? Если $a+b < n$, где n — порядок группы, то нам известно, что такой элемент находится в группе по определению. А если $a+b >= n$, то нам известно, что $a < n$ и $b < n$, а следовательно, $a+b < 2n$; значит, $a+b-n < n$:

$$(a + b - n)G = aG + bG - nG = aG + bG - 0 = aG + bG$$

В более широком смысле $(a + b)G = ((a + b) \% n)G$, где n — порядок группы. Таким образом, нам известно, что данный элемент находится в группе, обосновывая тем самым замкнутость.

Обратимость

Наглядно представить обратимость нетрудно (рис. 3.6).

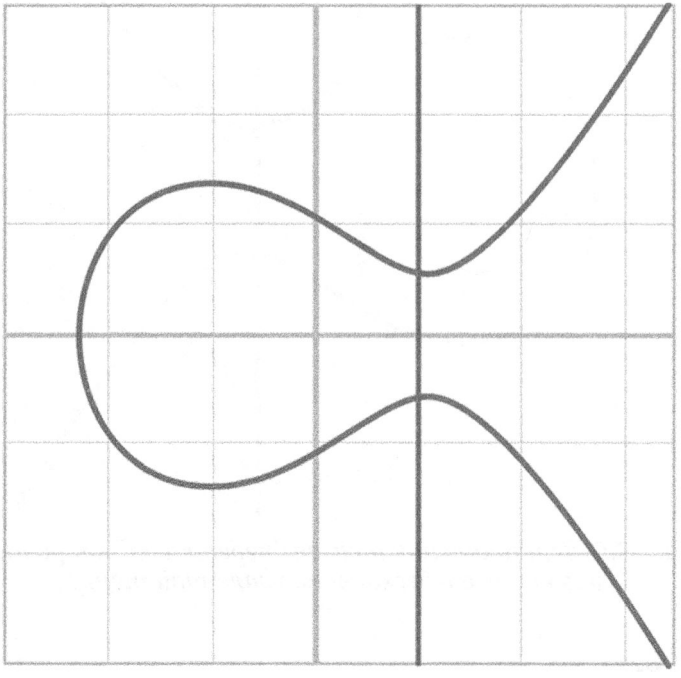

Рис. 3.6. Каждая точка на кривой обращается как зеркальное отображение относительно оси x

Если же рассматривать обратимость с математической точки зрения, то известно, что если элемент aG принадлежит группе, то ей принадлежит и элемент $(n - a)G$. Эти элементы можно сложить, чтобы получить в итоге следующий результат:

$$aG + (n - a)G = (a + n - a)G = nG = 0$$

Коммутативность

Из операции сложения точек известно, что $A + B = B + A$ (рис. 3.7).

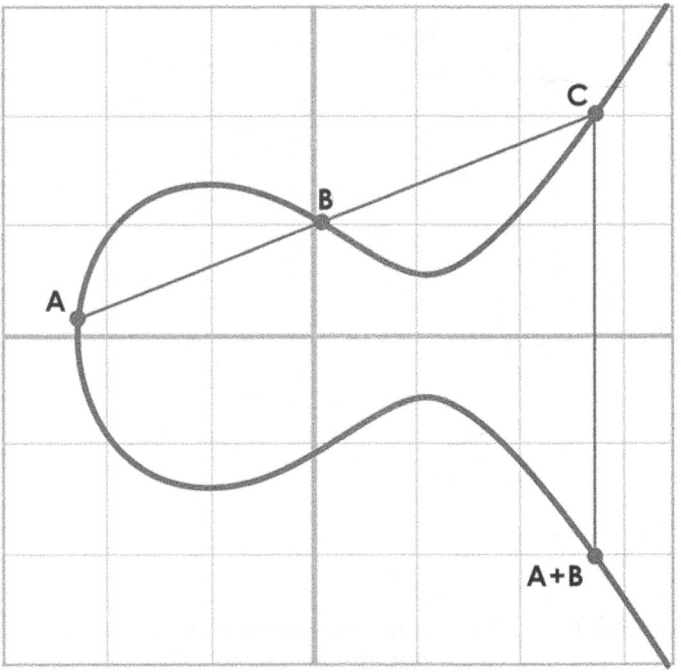

Рис. 3.7. Прямая линия, проведенная через точки, не меняется

Это означает, что $aG + bG = bG + aG$, а это и доказывает коммутативность.

Ассоциативность

Из операции сложения точек также известно, что $A + (B + C) = (A + B) + C$ (рис. 3.8 и 3.9).

Таким образом, $aG + (bG + cG) = (aG + bG) + cG$, что и доказывает ассоциативность.

Упражнение 5

Найдите порядок группы, сформированной из точки с координатами (15,86) для кривой, описываемой уравнением $y^2 = x^3 + 7$ над F_{223}.

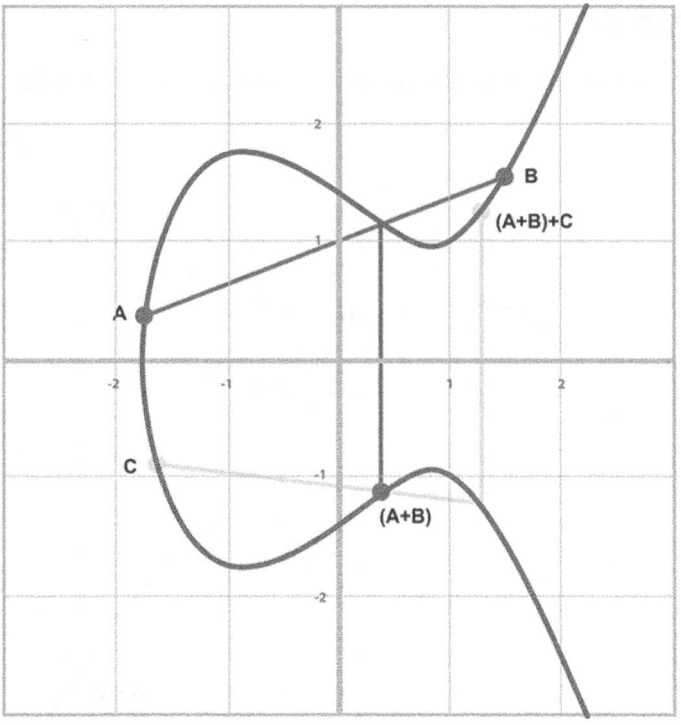

Рис. 3.8. (A + B) + C: сначала вычисляется сумма A + B, а затем прибавляется C

Программная реализация скалярного умножения

В упражнении 5 предпринята попытка выполнить следующие действия:

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x = FieldElement(15, prime)
>>> y = FieldElement(86, prime)
>>> p = Point(x, y, a, b)
>>> print(7*p)
Point(infinity)
```

В данном случае требуется скалярно умножить точку на некоторое число. Правда, в языке Python имеется метод `__rmul__()`, которым можно заменить операцию умножения на коэффициент. Наивная реализация такого метода выглядит следующим образом:

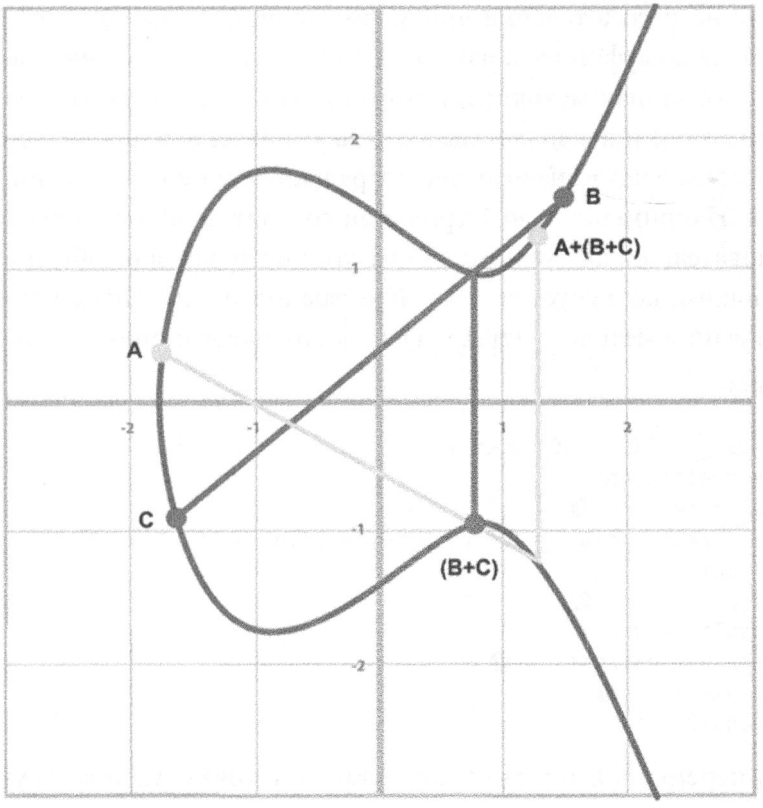

Рис. 3.9. A + (B + C): сначала вычисляется сумма B + C, а затем прибавляется A. В итоге получается та же самая точка, как и на рис. 3.8

```
class Point:
  ...
  def __rmul__(self, coefficient):
    product = self.__class__(None, None, self.a, self.b)     ❶
    for _ in range(coefficient):     ❷
      product += self
    return product
```

❶ Начальное значение переменной `product` равно нулю, что для сложения точек означает бесконечно удаленную точку.

❷ Цикл повторяется столько раз, сколько задано в параметре `coefficient`, и каждый раз прибавляется точка.

Такой реализации оказывается достаточно для небольших коэффициентов. Но что если коэффициент настолько большой, что выйти из цикла

умножения на него в течение приемлемого времени не удастся? Например, умножение на коэффициент, равный 1 триллиону, отнимет немало времени.

Имеется отличная методика, которая называется *двоичным разложением* (binary expansion) и позволяет выполнять умножение в циклах по логарифму $\log_2(n)$, благодаря чему значительно сокращается время вычисления для больших чисел. Например, число 1 триллион состоит из 40 бит в двоичной форме, а следовательно, для умножения на это число, которое обычно считается очень большим, потребуется всего 40 итераций цикла. Ниже приведена еще одна реализация метода `__rmul__()` по методике двоичного разложения.

```
class Point:
  ...
  def __rmul__(self, coefficient):
    coef = coefficient
    current = self     ❶
    result = self.__class__(None, None, self.a, self.b)     ❷
    while coef:
      if coef & 1:     ❸
        result += current
      current += current     ❹
      coef >>= 1     ❺
  return result
```

❶ Здесь переменная `current` представляет точку, умножаемую на текущий бит. На первой итерации цикла она представляет произведение `1 × self`; на второй — произведение `2 × self`; на третьей — произведение `4 × self`; на четвертой — произведение `8 × self` и т.д. Таким образом, на каждой итерации цикла количество точек удваивается. В двоичной форме это соответствует коэффициентам 1, 10, 100, 1000, 10 000 и т.д.

❷ Сначала результат скалярного умножения равен нулю, что соответствует бесконечно удаленной точке.

❸ Здесь проверяется, не равен ли младший бит (т.е. крайний справа) единице. Если он равен 1, то к результату прибавляется значение текущего бита.

❹ Количество точек необходимо удваивать до тех пор, пока не будет достигнута максимальная величина коэффициента.

❺ Здесь осуществляется поразрядный сдвиг коэффициента вправо.

Это непростая методика. И если вы не разбираетесь в принципе действия поразрядных операций, то представьте себе коэффициент в двоичной форме

и учтите, что точка прибавляется лишь в том случае, если в коэффициенте присутствуют единицы.

Итак, применяя методы `__add__()` и `__rmul__()`, мы можем приступить к определению более сложных эллиптических кривых.

Определение кривой для биткойна

И хотя мы пользовались относительно малыми простыми числами ради простоты рассмотренных ранее примеров, мы отнюдь ими не ограничиваемся. Малые простые числа означают, что мы можем осуществлять поиск в группе, используя компьютер. Так, если размер группы равен 301, то на компьютере можно без особого труда выполнить 301 вычисление, чтобы обратить скалярное умножение или решить задачу дискретного логарифмирования.

Но что если сделать простое число большим? Оказывается, что можно выбрать намного *б*ольшие простые числа, чем те, которыми мы пользовались раньше. Безопасность криптографии по эллиптическим кривым зависит от *неспособности* компьютеров преодолеть ощутимую долю группы.

Эллиптическая кривая для криптографии с открытым ключом определяется по следующим параметрам.

- В уравнении $y^2 = x^3 + ax + b$, описывающем кривую, определяются константы a и b.
- В качестве порядка конечного поля задается простое число p.
- Задаются координаты x и y генераторной точки G.
- Задается порядок n группы, формируемой из точки G.

Эти числовые параметры общеизвестны, а совместно они образуют криптографическую кривую. Имеется немало криптографических кривых, которым присущи разные компромиссы между безопасностью и удобством применения, но нас больше всего интересует кривая secp256k1, применяемая в биткойне. Ниже приведены параметры этой кривой.

- Константы $a = 0$, $b = 7$, приводящие к уравнению $y^2 = x^3 + 7$.
- $p = 2^{256} - 2^{32} - 977$.
- G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798.

- G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8.
- n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141.

Здесь G_x обозначает координату x, а G_y — координату y генераторной точки. Числа, начинающиеся с префикса 0x, являются шестнадцатеричными.

Данной кривой присущ ряд особенностей. Во-первых, ее описывает относительно простое уравнение. Константы a и b в уравнениях многих других кривых имеются намного бо́льшие значения.

Во-вторых, порядок p конечного поля очень близок к числу 2^{256}. Это означает, что большинство чисел, которые меньше 2^{256}, находятся в простом конечном поле, а следовательно, любая точка на кривой обладает координатами x и y, которые можно выразить 256 битами. Порядок n конечной группы также очень близок к числу 2^{256}. Это означает, что и любое скалярное кратное можно выразить 256 битами.

И в-третьих, число 2^{256} довольно большое, как поясняется ниже. Примечательно, что любое число, которое меньше 2^{256}, можно сохранить в 32 байтах. Это означает, что хранить секретный ключ относительно просто.

Насколько велико число 2^{256}

На первый взгляд, число 2^{256} не кажется большим, поскольку его можно выразить кратко. Но на самом деле это неимоверно большое число. Чтобы стала понятнее его величина, ниже приведены для примера некоторые сравнимые по масштабам числовые показатели.

$2^{256} \sim 10^{77}$

- Количество атомов на планете Земля: $\sim 10^{50}$.
- Количество атомов в Солнечной системе: $\sim 10^{57}$.
- Количество атомов в Млечном пути: $\sim 10^{68}$.
- Количество атомов во Вселенной: $\sim 10^{80}$.

Триллион (10^{12}) компьютеров, выполняющих триллион вычислений каждую триллионную (10^{-12}) долю секунды в течение триллиона лет, все равно произведут меньше, чем 10^{56} вычислений.

Если искать секретный ключ подобным способом, то следует учесть, что в биткойне столько возможных секретных ключей, сколько атомов в миллиардах галактик.

Работа с кривой secp256k1

Итак, зная все параметры эллиптической кривой secp256k1, можно создать код Python, проверяющий, находится ли генераторная точка G на кривой, описываемой уравнением $y^2 = x^3 + 7$, как показано ниже.

```
>>> gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
>>> gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
>>> p = 2**256 - 2**32 - 977
>>> print(gy**2 % p == (gx**3 + 7) % p)
True
```

Кроме того, код Python может проверить, имеет ли генераторная точка G порядок n:

```
>>> from ecc import FieldElement, Point
>>> gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28\
d959f2815b16f81798
>>> gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a685541\
99c47d08ffb10d4b8
>>> p = 2**256 - 2**32 - 977
>>> n = 0xfffffffffffffffffffffffffffffebaaedce6af48a03\
bbfd25e8cd0364141
>>> x = FieldElement(gx, p)
>>> y = FieldElement(gy, p)
>>> seven = FieldElement(7, p)
>>> zero = FieldElement(0, p)
>>> G = Point(x, y, zero, seven)
>>> print(n*G)
Point(infinity)
```

Теперь нам известна кривая, с которой нам предстоит работать, и поэтому настало время создать на языке Python подкласс для работы с эллиптической кривой secp256k1 по ее параметрам. Для этого определим объект, равнозначный объектам `FieldElement` и `Point`, но характерный для кривой secp256k1. Итак, начнем с определения того конечного поля, с которым нам предстоит работать, как показано ниже.

```
P = 2**256 - 2**32 - 977
...
class S256Field(FieldElement):

  def __init__(self, num, prime=None):
    super().__init__(num=num, prime=P)

    def __repr__(self):
      return '{:x}'.format(self.num).zfill(64)
```

Мы выполняем подклассификацию класса `FieldElement`, и потому нам не нужно всякий раз передавать параметр `P`. Кроме того, 256-разрядное число требуется отображать согласованно, и поэтому, дополняя выводимый результат до 64 знаков, мы можем наблюдать любые начальные нули.

Аналогично можно определить точку на кривой secp256k1 и присвоить ей имя `S256Point`:

```
A = 0
B = 7
...
class S256Point(Point):
  def __init__(self, x, y, a=None, b=None):
    a, b = S256Field(A), S256Field(B)
    if type(x) == int:
      super().__init__(x=S256Field(x), y=S256Field(y), a=a, b=b)
    else:
      super().__init__(x=x, y=y, a=a, b=b)     ❶
```

❶ Если инициализируется бесконечно удаленная точка, координаты x и y придется задать непосредственно, а не через класс `S256Field`.

Теперь нам легче инициализировать точку на кривой secp256k1, не определяя всякий раз параметры a и b, как это приходится делать в классе `Point`. Мы можем также определить метод `__rmul__()` более эффективно, поскольку нам известен порядок n конечной группы. И поскольку мы программируем на языке Python, обозначим этот порядок прописной буквой `N`, чтобы стало понятно, что это константа, как показано ниже.

```
N = 0xfffffffffffffffffffffffffffffebaaedce6af48\
a03bbfd25e8cd0364141
...
class S256Point(Point):
  ...
  def __rmul__(self, coefficient):
    coef = coefficient % N      ❶
    return super().__rmul__(coef)
```

❶ Здесь можно выполнить операцию по модулю n, поскольку $nG = 0$. Это означает, что через каждые n итераций цикла мы возвращаемся к нулю или к бесконечно удаленной точке.

Точку G теперь можно определить непосредственно и сохранить ее, поскольку мы будем еще не раз пользоваться ею в дальнейшем:

```
G = S256Point(
  0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
  0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
```

Проверить порядок n точки G теперь не составит большого труда:

```
>>> from ecc import G, N
>>> print(N*G)
S256Point(infinity)
```

Криптография с открытым ключом

Наконец-то, в нашем распоряжении имеются все инструментальные средства, требующиеся для выполнения операций криптографии с открытым ключом. Главной из них является операция, описываемая асимметричным уравнением $P = eG$. Вычислить P нетрудно, если известны e и G, но не так-то просто вычислить e, даже если известны P и G. В этом, собственно, и состоит упоминавшаяся ранее задача дискретного логарифмирования.

Понимание трудностей дискретного логарифмирования имеет существенное значение для уяснения алгоритмов подписания и верификации.

В общем, e называется *секретным ключом* (private key), а P — *открытым ключом* (public key). Следует, однако, иметь в виду, что секретный ключ представлен 256-разрядным числом, а открытый ключ — координатами (x,y), где x и y — также 256-разрядные числа.

Подписание и верификация

Чтобы обосновать причины, по которым существуют подписание и верификация, рассмотрим следующую ситуацию. Допустим, вам требуется доказать, что вы действительно хороший лучник и способны попасть в любую, а не конкретную цель на расстоянии 50 метров.

Если кто-нибудь будет наблюдать за вами и общаться с вами, доказать свои способности лучника вам будет нетрудно. Возможно, для проверки ваших способностей кому-то придет в голову поставить своего сына с яблоком в руке на расстоянии 40 м и предложить вам попасть в яблоко стрелой. Если вы действительно хороший лучник, то сможете подтвердить свою квалификацию. А мишень, указанная вашим оппонентом, позволит легко проверить ваши навыки стрельбы из лука.

Но, к сожалению, такой способ доказательства не особенно поддается нормированию. Так, если вам потребуется доказать свои способности лучника 10 людям, вам придется выпустить из лука 10 стрел в 10 разных мишеней с 10 разных расстояний. И хотя вы можете попытаться собрать 10 людей вместе, чтобы они наблюдали за тем, как вы стреляете из лука в одну конкретную мишень, вы не убедите их, что сможете с таким же успехом попасть в произвольную цель, поскольку они все сразу не смогут выбрать такую цель. Следовательно, требуется найти такой способ, который позволил бы вам проявить свои способности один раз, а не иметь всякий раз дело с разными проверяющими, но в то же время доказать, что вы действительно хороший лучник и можете попасть в любую цель.

Так, если вы просто стреляете из лука в выбранную вами мишень, то совсем не обязательно убедите тех, кто будет наблюдать впоследствии результаты вашей стрельбы. Ведь вы могли нарисовать мишень вокруг того места, куда попала стрела. Что же тогда вам делать?

Вы можете поступить весьма благоразумно, надписав наконечник стрелы положением мишени, в которую вы попадаете из лука (например, "яблоко на голове моего сына"), а затем попасть в эту мишень данной стрелой. И тогда любой, кто осматривает мишень, может произвести рентгеноскопию наконечника данной стрелы и убедиться, что на ней действительно написана именно та мишень, в которую она попала. А поскольку наконечник стрелы был ясно надписан перед выстрелом в мишень, данное обстоятельство может стать доказательством того, что вы действительно хороший лучник, если, конечно, это не была мишень, попадать в которую вы упражнялись неоднократно.

Аналогичная методика применяется при подписании и верификации, за исключением того, что в данном случае доказываются не способности хорошего лучника, а знание секретного числа. Нам, в частности, требуется доказать, что мы обладаем секретом, не раскрывая его. И для этого мы вводим цель в свое вычисление и попадаем в нее. В конечном счете такая методика применяется в транзакциях для подтверждения того, что биткойны расходуют законные владельцы.

Надписание цели

Надписание цели зависит от *алгоритма подписи* (signature algorithm), и в данном случае такой алгоритм называется "ECDSA" (Elliptic Curve Digital

Signature Algorithm — алгоритм цифровых подписей на основе эллиптических кривых).

Секретом в данном случае является число e, удовлетворяющее следующему условию:

$$eG = P$$

Здесь P — открытый ключ, а e — секретный ключ.

Цель, в которую мы собираемся попасть, является случайным 256-разрядным числом k. И для этого мы должны сделать следующее:

$$kG = R$$

Теперь R является целью, в которую мы собираемся попасть. В действительности нас интересует лишь координата x цели R, которую мы обозначим как r, т.е. *случайное* число.

В данный момент мы можем утверждать, что следующее уравнение равнозначно задаче дискретного логарифмирования:

$$uG + vP = kG$$

Здесь k выбрано произвольно, u, $v \neq 0$ и могут быть выбраны подписавшей стороной, а P и G известно благодаря тому, что

$$\text{из } uG + vP = kG \text{ следует, что } vP = (k - u)G$$

А поскольку $v \neq 0$, можно разделить на скалярное кратное v:

$$P = ((k - u)/v)G$$

Если e известно, то мы имеем следующее:

$$eG = ((k - u)/v)G \text{ или } e = (k - u)/v$$

Это означает, что любого сочетания (u,v), удовлетворяющего предыдущему уравнению, окажется достаточно.

Если же e неизвестно, то придется экспериментировать с сочетанием (u,v) до тех пор, пока не будет удовлетворено уравнение $e = (k - u)/v$. Если бы его можно было решить при любом сочетании (u,v), это означало бы, что нам удалось решить и уравнение $P = eG$, зная только P и G. Иными словами, мы решили бы задачу дискретного логарифмирования.

Это означает следующее: чтобы обеспечить правильность величин u и v, придется решить задачу дискретного логарифмирования, а иначе необходимо знать секретную величину e. Но поскольку мы допускаем, что решить задачу

дискретного логарифмирования трудно, вполне можно допустить, что секретная величина e известна тому, кто предложил величины u и v.

Еще один невыясненный до сих пор вопрос — как внедрить цель нашей стрельбы. Речь, по существу, идет о договоре, который должен быть выполнен в результате стрельбы по мишени. Например, Вильгельм Телль стрелял настолько искусно, что сумел спасти своего сына (почаще стреляйте по мишени, чтобы спасти своего сына). Нетрудно себе представить, что для попадания в цель могут быть другие причины и "награды", которые может завоевать стрелок. И все это придется внедрить в рассматриваемые здесь уравнения.

В терминологии подписания и верификации это называется *хешем подписи*, где хеш — это детерминированная функция, получающая произвольные данные и преобразующая их в данные фиксированной величины. Эти данные, по существу, являются отпечатком сообщения, содержащего намерения стрелка, которые уже известны всем, кто проверяет полученное сообщение. Обозначим эти данные буквой z и внедрим их в расчет упомянутой выше суммы $uG + vP$ следующим образом:

$$u = z/s,\ v = r/s$$

Поскольку r используется в вычислении величины v, мы тем самым надписали наконечник стрелы. Мы также внедрили намерение стрелка непосредственно в уравнение, определяющее величину u, а потому и причина для стрельбы и ее цель теперь входят в данное уравнение.

Чтобы привести уравнение $uG + vP = kG$ к пригодному для применения виду, рассчитаем величину s приведенным ниже образом. Полученный в итоге результат послужит основанием для алгоритма подписи, где r и s — два числа в подписи.

$$uG + vP = R = kG$$
$$uG + veG = kG$$
$$u + ve = k$$
$$z/s + r/s = k$$
$$(z + re)/s = k$$
$$s = (z + re)/k$$

А верификация реализуется довольно просто, как показано ниже.

$uG + vP$, где $u, v \neq 0$

$$uG + vP = (z/s)G + (re/s)G = ((z + re)/s)G = ((z + re)/((z + re)/k))G = \\ = kG = (r,y)$$

Почему не раскрывается величина *k*

В данный момент у вас может возникнуть вполне резонный вопрос: почему мы раскрываем не величину *k*, а координату *x* цели *R*, т.е. величину *r*? Если бы мы раскрыли величину *k*, то получили бы приведенное ниже соотношение, которое означает, что наш секрет раскрыт, а следовательно, назначение подписи лишено всякого смысла. Но мы можем раскрыть цель *R*.

$$uG + vP = R$$
$$uG + veG = kG$$
$$kG - uG = veG$$
$$(k - u)G = veG$$
$$(k - u) = ve$$
$$(k - u)/v = e$$

Стоит еще раз напомнить, что в качестве величины *k* следует непременно употреблять истинно случайные числа. Ведь даже случайное раскрытие величины *k* для известной подписи равнозначно раскрытию секрета и потере собственных денежных средств!

Подробнее о верификации

Подписи служат для подписания некоторого значения фиксированной длины (например, 32 байта). В рассматриваемом здесь случае этим значением является упомянутый выше "договор". А то, что 32 байта состоят из 256 битов, не является случайным совпадением, поскольку подписываемое значение служит скалярным кратным для точки *G*.

Для того чтобы длина подписываемого документа непременно составила 32 байта, необходимо сначала хешировать этот документ. Этим гарантируется, что будет подписан документ длиной ровно 32 байта. Для этого в биткойне применяется функция хеширования hash256 или выполненная за два цикла функция хеширования sha256. Этим также гарантируется, что будет подписан документ длиной ровно 32 байта. Результат хеширования называется *хешем подписи* (signature hash) и обозначается как *z*.

Подпись, проверяемая при верификации, образуется из двух составляющих (*r*,*s*), где *r* — координата *x* некоторой целевой точки *R*, к которой мы еще вернемся. А формула для расчета величины *s* такая же, как и выведенная выше:

$s = (z + re)/k$

В этой формуле нам известно e (из уравнения $P = eG$, иначе то, что мы доказываем, должно быть нам известно в первую очередь). Нам также известны k (из выведенного ранее уравнения $kG = R$) и z.

Составим теперь уравнение $R = uG + vP$, определив u и v как

$u = z/s$
$v = r/s$

Таким образом, получаем

$$uG + vP = (z/s)G + (r/s)P = (z/s)G + (re/s)G = ((z + re)/s)G$$

Нам уже известно, что $s = (z + re)/k$, а следовательно

$$uG + vP = ((z + re) / ((z + re)/k))G = kG = R$$

Итак, мы удачно выбрали величины u и v таким образом, чтобы сформировать целевую точку R, как и намеревались сделать. Более того, мы использовали r при вычислении величины v, подтвердив тем самым, что нам было заранее известно, какой будет целевая точка R. Но знать заранее о целевой точке R можно лишь в том случае, если известно e.

Таким образом, чтобы сформировать целевую точку R, необходимо выполнить следующие действия.

1. Исходно задать подпись в виде (r,s), z — в виде хеша подписываемого документа, а P — в виде открытого ключа (или открытой точки) подписавшей стороны.
2. Вычислить $u = z/s$, $v = r/s$.
3. Вычислить $uG + vP = R$.
4. Если координата x целевой точки R равна r, то подпись действительна.

Зачем выполнять функцию хеширования sha256 за два цикла

Чтобы вычислить величину z, необходимо выполнить функцию хеширования hash256 или же функцию хеширования sha256 за два цикла. В связи с этим может возникнуть вопрос: зачем нужны два цикла, если требуется всего лишь получить 256-разрядное число? Все дело в безопасности.

Хорошо известна атака на основе коллизии хеш-функций в алгоритме шифрования SHA-1, иначе называемая *атакой типа "дней рождения"*, которая упрощает поиск подобных коллизий. В компании Google такая коллизия была обнаружена в 2017 году с помощью некоторых модификаций атаки типа "дней рождения" и многих других приемов (`https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`). Поэтому *двойное* шифрование по алгоритму SHA-1 служит мерой борьбы или хотя бы замедления некоторых форм такой атаки.

И хотя выполнение функции хеширования sha256 за два цикла совсем необязательно предотвращает все возможные виды атак, такой прием защищает от потенциальных уязвимостей.

Верификация подписи

А теперь можно произвести верификацию подписи, используя некоторые из имеющихся в нашем распоряжении криптографических примитивов, как показано ниже.

```
>>> from ecc import S256Point, G, N
>>> z = 0xbc62d4b80d9e36da29c16c5d4d9f11731f36052c72401\
a76c23c0fb5a9b74423
>>> r = 0x37206a0610995c58074999cb9767b87af4c4978db68\
c06e8e6e81d282047a7c6
>>> s = 0x8ca63759c1157ebeaec0d03cecca119fc9a75bf8e6\
d0fa65c841c8e2738cdaec
>>> px = 0x04519fac3d910ca7e7138f7013706f619fa8f033\
e6ec6e09370ea38cee6a7574
>>> py = 0x82b51eab8c27c66e26c858a079bcdf4f1ada34ce\
c420cafc7eac1a42216fb6c4
>>> point = S256Point(px, py)
>>> s_inv = pow(s, N-2, N)     ❶
>>> u = z * s_inv % N     ❷
>>> v = r * s_inv % N     ❸
>>> print((u*G + v*point).x.num == r)     ❹
True
```

❶ Для вычисления $1/s$ здесь применяется малая теорема Ферма, поскольку n — простое число.

❷ Вычислить $u = z/s$.

❸ Вычислить $v = r/s$.

❹ Вычислить $uG + vP = (r,y)$. Здесь необходимо проверить, что координата x равна r.

Упражнение 6

Проверьте, действительны ли следующие подписи:

```
P = (0x887387e452b8eacc4acfde10d9aaf7f6d9a0f975aabb10d006e4da568744d06c,
     0x61de6d95231cd89026e286df3b6ae4a894a3378e393e93a0f45b666329a0ae34)

# подпись 1
z = 0xec208baa0fc1c19f708a9ca96fdeff3ac3f230bb4a7ba4aede4942ad003c0f60
r = 0xac8d1c87e51d0d441be8b3dd5b05c8795b48875dffe00b7ffcfac23010d3a395
s = 0x68342ceff8935ededd102dd876ffd6ba72d6a427a3edb13d26eb0781cb423c4

# подпись 2
z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d
r = 0xeff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c
s = 0xc7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6
```

Программная реализация верификации подписей

У нас уже имеется класс `S256Point`, представляющий открытую точку для секретного ключа, поэтому создадим класс `Signature` для хранения величин r и s, как показано ниже. Мы еще вернемся к этому классу в главе 4, чтобы расширить его функциональные возможности.

```
class Signature:

  def __init__(self, r, s):
    self.r = r
    self.s = s

  def __repr__(self):
    return 'Signature({:x},{:x})'.format(self.r, self.s)
```

И на этом основании можно теперь определить метод `verify()` в классе `S256Point` следующим образом:

```
class S256Point(Point):
  ...
  def verify(self, z, sig):
    s_inv = pow(sig.s, N - 2, N)     ❶
    u = z * s_inv % N     ❷
    v = sig.r * s_inv % N     ❸
```

```
    total = u * G + v * self     ❹
    return total.x.num == sig.r     ❺
```

❶ Для вычисления $1/s$ здесь применяется малая теорема Ферма, поскольку порядок n конечной группы — простое число.

❷ Вычислить $u = z/s$. Следует иметь в виду, что здесь можно выполнить операцию по модулю n, поскольку n — порядок группы.

❸ Вычислить $v = r/s$. Следует иметь в виду, что здесь можно выполнить операцию по модулю n, поскольку n — порядок группы.

❹ Вычислить сумму $uG + vP$, которая должна быть равна R.

❺ Проверить координату x на равенство r.

Итак, имея в своем распоряжении открытый ключ в виде точки на эллиптической кривой secp256k1, а также хеш подписи (z), мы можем проверить, действительна ли подпись.

Подробнее о подписании

Если знать, каким образом должна выполняться верификация, то нетрудно осуществить и подписание. Остается лишь выяснить, какую величину k, а следовательно, $R = kG$, необходимо выбрать. С этой целью выберем случайную величину k.

Итак, процедура подписания осуществляется следующим образом.

1. Исходно задана величина z, а также известна величина e, так что $eG = P$.
2. Выбрать случайную величину k.
3. Вычислить $R = kG$ и r = координата x целевой точки R.
4. Вычислить $s = (z + re)/k$.
5. Таким образом, подпись создана в виде (r,s).

Следует, однако, иметь в виду, что открытый ключ P должен быть передан тому, кому требуется произвести его верификацию и для этого ему необходимо также знать величину z. Как будет показано далее, величина z вычисляется, а открытый ключ P отсылается вместе с подписью.

Создание подписи

Теперь мы можем создать подпись.

Внимательно относитесь к созданию случайных чисел

Имейте в виду, что пользоваться чем-то вроде имеющейся в Python библиотеки `random` для целей криптографии *не* рекомендуется. Эта библиотека, в частности, предназначена для применения только в учебных целях, поэтому ни в коем случае не пользуйтесь описываемым здесь кодом для производственных целей.

Для этого воспользуемся некоторыми из имеющихся в нашем распоряжении криптографических примитивов.

```
>>> from ecc import S256Point, G, N
>>> from helper import hash256
>>> e = int.from_bytes(hash256(b'my secret'), 'big')     ❶
>>> z = int.from_bytes(hash256(b'my message'), 'big')    ❷
>>> k = 1234567890     ❸
>>> r = (k*G).x.num    ❹
>>> k_inv = pow(k, N-2, N)
>>> s = (z+r*e) * k_inv % N     ❺
>>> point = e*G     ❻
>>> print(point)
S256Point(028d003eab2e428d11983f3e97c3fa0addf3b42740df0d211795ffb3be2f6c52, \
0ae987b9ec6ea159c78cb2a937ed89096fb218d9e7594f02b547526d8cd309e2)
>>> print(hex(z))
0x231c6f3d980a6b0fb7152f85cee7eb52bf92433d9919b9c5218cb08e79cce78
>>> print(hex(r))
0x2b698a0f0a4041b77e63488ad48c23e8e8838dd1fb7520408b121697b782ef22
>>> print(hex(s))
0xbb14e602ef9e3f872e25fad328466b34e6734b7a0fcd58b1eb635447ffae8cb9
```

❶ Это пример "кошелька в уме", предназначенного для хранения секретного ключа в своем уме и облегчения его запоминания. Впрочем, пользоваться подобным средством для хранения настоящих секретов не рекомендуется.

❷ Это хеш подписи или хеш подписываемого сообщения.

❸ Здесь предполагается использовать фиксированную величину k для демонстрационных целей.

❹ $kG = (r,y)$, поэтому здесь взята лишь координата x.

❺ $s = (z + re)/k$. Здесь можно выполнить операцию по модулю n, поскольку известно, что n — это порядок конечной циклической группы.

❻ Открытая точка должна быть известна тому, кто производит верификацию.

Упражнение 7

Подпишите следующее сообщение заданным секретным ключом:

```
e = 12345
z = int.from_bytes(hash256('Programming Bitcoin!'), 'big')
```

Программная реализация подписания сообщений

Для программной реализации подписания сообщений, необходимо создать класс `PrivateKey`, в объекте которого будет храниться секретная информация, как показано ниже.

```
class PrivateKey:

  def __init__(self, secret):
    self.secret = secret
    self.point = secret * G

  def hex(self):
    return '{:x}'.format(self.secret).zfill(64)     ❶
```

❶ Здесь ради удобства сохраняется открытый ключ (`self.point`).

Затем необходимо определить метод `sign()` следующим образом:

```
from random import randint
...
class PrivateKey:
...
  def sign(self, z):
    k = randint(0, N)     ❶
    r = (k*G).x.num       ❷
    k_inv = pow(k, N-2, N)     ❸
    s = (z + r*self.secret) * k_inv % N     ❹
    if s > N/2:     ❺
    s = N - s
    return Signature(r, s)     ❻
```

❶ В функции `randint()` выбирается произвольное целое число, находящееся в пределах от 0 до n. Но в реальных случаях пользоваться этой функцией из библиотеки `random` *не* рекомендуется, поскольку она возвращает число, которое в итоге оказывается недостаточно случайным.

❷ Вычислить r — координату x целевой точки R.

❸ И здесь применяется малая теорема Ферма, поскольку порядок n конечной группы — простое число.

❹ Вычислить уравнение $s = (z + re)/k$.

❺ При малой величине s получаются узлы для передачи транзакций. Это делается для большей гибкости.

❻ Здесь возвращается объект определенного ранее класса `Signature`.

Важность уникальности значения *k*

Подписи подчиняются очень важному правилу, определяющему случайную составляющую (в данном случае — k), величина которой должна быть непременно уникальной для каждой подписи. Это означает, что величина k не может использоваться повторно. Ведь если величина k используется повторно, секрет раскрывается! Почему? Потому, что если имеется секрет e, а величина k используется повторно для подписания сообщений z_1 и z_2, то всякий, кто обнаруживает обе подписи, может раскрыть секрет по выведенной ниже формуле! Так, система защиты игровой приставки PlayStation 3 была взломана в 2010 году из-за того, что величина k использовалась в нескольких подписях (`https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/`).

$kG = (r,y)$

$s_1 = (z_1 + re) / k, s_2 = (z_2 + re) / k$

$s_1/s_2 = (z_1 + re) / (z_2 + re)$

$s_1(z_2 + re) = s_2(z_1 + re)$

$s_1z_2 + s_1re = s_2z_1 + s_2re$

$s_1re - s_2re = s_2z_1 - s_1z_2$

$e = (s_2z_1 - s_1z_2) / (rs_1 - rs_2)$

В качестве меры борьбы с раскрытием секретов выработан стандарт на создание каждый раз уникальной детерминированной величины k исходя из имеющегося секрета и подписываемого сообщения z. Спецификация этого стандарта приведена в документе RFC 6979 (`https://tools.ietf.org/html/rfc6979`), а соответствующие изменения в исходном коде выглядят следующим образом:

```
class PrivateKey:
...
  def sign(self, z):
    k = self.deterministic_k(z)      ❶
    r = (k * G).x.num
```

```
    k_inv = pow(k, N - 2, N)
    s = (z + r * self.secret) * k_inv % N
    if s > N / 2:
      s = N - s
    return Signature(r, s)

  def deterministic_k(self, z):
    k = b'\x00' * 32
    v = b'\x01' * 32
    if z > N:
      z -= N
    z_bytes = z.to_bytes(32, 'big')
    secret_bytes = self.secret.to_bytes(32, 'big')
    s256 = hashlib.sha256
    k = hmac.new(k, v + b'\x00' + secret_bytes
                 + z_bytes, s256).digest()
    v = hmac.new(k, v, s256).digest()
    k = hmac.new(k, v + b'\x01' + secret_bytes
                 + z_bytes, s256).digest()
    v = hmac.new(k, v, s256).digest()
    while True:
      v = hmac.new(k, v, s256).digest()
      candidate = int.from_bytes(v, 'big')
      if candidate >= 1 and candidate < N:
        return candidate      ❷
      k = hmac.new(k, v + b'\x00', s256).digest()
      v = hmac.new(k, v, s256).digest()
```

❶ Здесь используется детерминированная, а не случайная величина *k*. Все остальное тело метода `sign()` остается таким же, как и прежде.

❷ Этот алгоритм возвращает подходящего кандидата на уникальную величину *k*.

Таким образом, величина *k* оказывается уникальной с достаточно высокой степенью вероятности. И это становится возможным потому, что функция хеширования sha256 устойчива к коллизиям, которые до сих пор не обнаружены.

Еще одно преимущество такого подхода, с точки зрения тестирования, заключается в том, что подпись заданного сообщения *z* и секретный ключ

остаются каждый раз неизменными. Благодаря этому значительно упрощается отладка исходного кода и написание модульных тестов. Кроме того, те транзакции, в которых применяется детерминированная величина k, каждый раз будут создаваться такими же самыми, поскольку не изменяется их подпись. И благодаря этому транзакции становятся менее податливыми (подробнее об этом — в главе 13).

Заключение

В этой главе были изложены основы криптографии по эллиптическим кривым, и это дает вам право утверждать, что вы теперь знаете, как хранить секрет, подписывая сообщение, и проверять подлинность того, кто фактически подписал сообщение. И даже если вы не прочтете больше ни одной страницы этой книги, вы все равно уже научились тому, что раньше называлось "оружием специального назначения" (`https://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States`). Таким образом, вы совершили главный шаг на пути к изучению рассматриваемого здесь предмета, что очень важно для усвоения материала остальной части этой книги!

А в следующей главе мы перейдем к обсуждению вопросов сериализации многих из рассмотренных здесь структур данных, чтобы хранить их на жестком диске и передавать по сети.

ГЛАВА 4

Сериализация

В предыдущих главах мы создали немало классов, в том числе `PrivateKey`, `S256Point` и `Signature`. А теперь мы должны подумать о том, как передать объекты этих классов на другие компьютеры в сети или хотя бы перенести на жесткий диск. Именно здесь и вступает в действие сериализация, поскольку нам требуется передать по сети или сохранить на жестком диске объект типа `S256Point`, `Signature` или `PrivateKey`. В идеальном случае желательно сделать это эффективно по причинам, объясняемым в главе 10.

Несжатый формат SEC

Итак, начнем с класса `S256Point`, представляющего открытый ключ. Напомним, что открытый ключ в криптографии по эллиптическим кривым на самом деле является целевой точкой с координатами (x,y). Как же сериализировать эти данные?

Оказывается, что для сериализации открытых ключей, формируемых по алгоритму ECDSA, уже имеется стандартный формат SEC (Standards for Efficient Cryptography — *Стандарты для эффективного шифрования*), причем слово "эффективного" в его названии предполагает минимальные издержки на шифрование. Формат SEC существует в двух формах: несжатой и сжатой. Начнем его рассмотрение с первой формы, а ко второй перейдем в следующем разделе.

Данные формируются в несжатом формате SEC для заданной точки $P = (x,y)$ в следующем порядке.

1. Сначала задается префиксный байт `0x04`.
2. Затем к нему присоединяются 32 байта, следующие в обратном порядке и обозначающие целочисленное значение координаты x.

3. Далее присоединяются еще 32 байта, следующие в обратном порядке и обозначающие целочисленное значение координаты *y*.

Структура несжатого формата SEC наглядно представлена на рис. 4.1.

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

        - 04 - Маркер
        - координата x — 32 байта
        - координата y — 32 байта
```

Рис. 4.1. Несжатый формат SEC

Прямой и обратный порядок следования байтов

Существенной побудительной причиной для кодировок с прямым и обратным порядком следования байтов служит хранение числовых данных на жестком диске. В частности, закодировать число, которое меньше 256, совсем не трудно, поскольку для его хранения на жестком диске достаточно единственного байта (2^8). Но если число больше 256, то как его сериализировать в последовательность байтов?

Арабские числа (в десятичной системе счисления) читаются слева направо. Так, число 123 читается как 100 + 20 + 3 (т.е. сто двадцать три), а не как 1 + 20 + 300. А в двоичной форме это число представлено в *обратном* порядке следования байтов от старшего к младшему, поскольку оно начинается с разряда сотен и оканчивается разрядом единиц. Но в вычислительной технике более эффективным иногда оказывается противоположный, *прямой* порядок следования байтов от младших к старшим.

А поскольку компьютеры обрабатывают данные в байтах, состоящих из 8 битов, десятичные числа приходится представлять в системе счисления по основанию 256. Это, например, означает, что десятичное число `500` может быть представлено как `01f4` в обратном порядке следования байтов, т.е. `500 = 1 × 256 + 244` (`f4` — в шестнадцатеричной форме), а в прямом следования байтов — как `f401`.

К сожалению, некоторые виды сериализации в биткойне (например, координат *x* и *y* в формате SEC) выполняются в обратном

порядке следования байтов, тогда как другие ее виды (например, номер версии транзакции, см. главу 5) — в прямом порядке следования байтов. В этой книге специально обращается внимание, когда сериализация данных выполняется в обратном порядке следования байтов, а когда — в прямом порядке.

Произвести сериализацию данных в несжатом формате SEC совсем не трудно. Труднее преобразовать 256-разрядное число в 32 байта, которые следуют в обратном порядке. Ниже показано, как это делается непосредственно в коде.

```
class S256Point(Point):
...
  def sec(self):
    '''Возвращает двоичный вариант данных формата SEC'''
    return b'\x04' + self.x.num.to_bytes(32, 'big') \
        + self.y.num.to_bytes(32, 'big')    ❶
```

❶ Начиная с версии Python 3 заданное число можно преобразовать в байты с помощью метода `to_bytes()`. В качестве первого аргумента при вызове этого метода указывается количество байтов для представления заданного числа, а качестве второго аргумента — порядок следования байтов (см. примечание приведенное выше).

Упражнение 1

Представьте открытый ключ в несжатом формате SEC, если имеются следующие секретные ключи.

- 5000
- 2018^5
- 0xdeadbeef12345

Сжатый формат SEC

Напомним, что любой координате x соответствуют по крайней мере две координаты y, что обусловлено наличием члена y^2 в уравнении, описывающем эллиптическую кривую (рис. 4.2). Оказывается, что такая же самая симметрия наблюдается и над конечным полем.

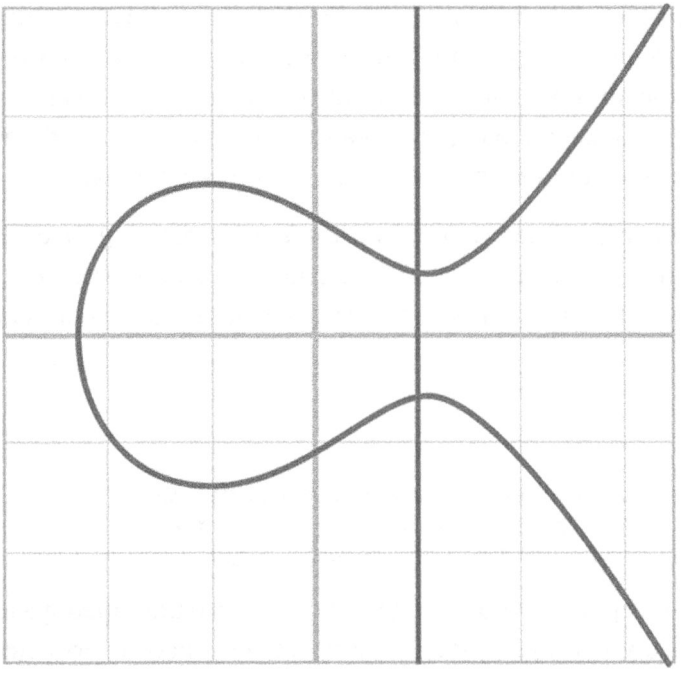

Рис. 4.2. Два возможных значения координаты y образуются в тех точках, где вертикальная линия пересекает эллиптическую кривую

Такая симметрия объясняется тем, что уравнению $y^2 = x^3 + ax + b$ удовлетворяет не только любая точка с координатами (x,y), но и точка с координатами $(x,-y)$. Более того, в конечном поле справедливо соотношение $-y \% p = (p - y) \% p$. А точнее, если точка с координатами (x,y) удовлетворяет уравнению, описывающему эллиптические координаты, то ему удовлетворяет и точка с координатами $(x,p - y)$. И это лишь два решения данного уравнения для заданной координаты x, как показано выше. Так, если известна координата x, то известно, что координата y может быть представлена как y или $p - y$.

А поскольку p — простое число, которое больше 2, известно, что оно нечетное. Так, если координата y представлена четным числом, то разность $p - y$ (т.е. нечетное число минус четное) дает в итоге нечетное число. А если координата y представлена нечетным числом, то разность $p - y$ дает в итоге четное число. Иными словами, число, находящееся в пределах от y до $p - y$, может быть либо четным, либо нечетным. И этим обстоятельством можно воспользоваться, чтобы сократить несжатый формат SEC, предоставив координату x и четность координаты y. Полученный в итоге формат SEC называется

сжатым потому, что координата y сжимается в единственный байт, каким бы числом она ни была представлена: четным или нечетным.

Ниже приведен порядок сериализации данных в сжатом формате SEC для заданной точки $P = (x,y)$.

1. Сначала задается префиксный байт. Если координата y представлена четным числом, то устанавливается префиксный байт `0x02`, а в противном случае — `0x03`.
2. Затем к нему присоединяются 32 байта, следующие в обратном порядке и обозначающие целочисленное значение координаты x.

Структура сжатого формата SEC наглядно представлена на рис. 4.3.

Рис. 4.3. Сжатый формат SEC

И эта процедура довольно проста. Для обработки ключей в сжатом формате SEC можно обновить метод `sec()` следующим образом:

```
class S256Point(Point):
...
  def sec(self, compressed=True):
    '''Возвращает двоичный вариант данных формата SEC'''
    if compressed:
      if self.y.num % 2 == 0:
        return b'\x02' + self.x.num.to_bytes(32, 'big')
      else:
        return b'\x03' + self.x.num.to_bytes(32, 'big')
    else:
      return b'\x04' + self.x.num.to_bytes(32, 'big') \
          + self.y.num.to_bytes(32, 'big')
```

Главное преимущество сжатого формата SEC заключается в том, что для хранения данных в этом формате требуется 33 байта, а не 65 байтов. И это немалая экономия, когда дело доходит до обработки миллионов транзакций.

В данный момент может возникнуть вопрос: как аналитически рассчитать координату y, если известна координата x? Для этого придется вычислить квадратный корень в конечном поле. Математически эту операцию можно сформулировать следующим образом:

Найти w из соотношения $w^2 = v$, если v известно.

Оказывается, что произвести такой расчет совсем не трудно, если известно, что порядок конечного поля определяется простым числом: p % 4 = 3. Ниже показано, как это делается.

Изначально известно, что

$$p \% 4 = 3$$

из этого следует, что

$$(p + 1) \% 4 = 0$$

То есть деление $(p + 1)/4$ дает целое значение.

По определению

$$w^2 = v$$

Нам же требуется вывести формулу для расчета w. Из малой теоремы Ферма известно, что

$$w^{p-1} \% p = 1$$

Это означает, что

$$w^2 = w^2 \times 1 = w^2 \times w^{p-1} = w^{(p+1)}$$

А поскольку p — нечетное число (напомним, что p — простое число, определяющее порядок конечного поля), известно, что даже если разделить $(p + 1)$ на 2, то все равно получится целое число. Исходя из этого, можно вывести следующую формулу:

$$w = w^{(p+1)/2}$$

А теперь можно воспользоваться целочисленным результатом деления $(p + 1)/4$ следующим образом:

$$w = w^{(p+1)/2} = w^{2(p+1)/4} = (w^2)^{(p+1)/4} = v^{(p+1)/4}$$

Таким образом, формула для нахождения квадратного корня принимает следующий вид:

если $w^2 = v$ и p % 4 = 3, то $w = v^{(p+1)/4}$

Оказывается, что в эллиптической кривой secp256k1 величина p используется таким образом, чтобы p % 4 == 3, что позволяет воспользоваться приведенными ниже формулами.

$$w^2 = v$$
$$w = v^{(p+1)/4}$$

В итоге будет получено одно из возможных значений w, а другим будет $p - w$. Такой итог объясняется тем, что для извлечения квадратного корня подходит как положительное, как и отрицательное числа.

Для извлечения квадратного корня по выведенной выше формуле непосредственно в коде можно ввести в класс `S256Field` следующий метод:

```
class S256Field(FieldElement):
...
  def sqrt(self):
    return self**((P + 1) // 4)
```

Для получения открытого ключа в формате SEC необходим метод `parse()`, определяющий требующуюся координату y. Ниже приведено определение этого метода в классе `S256Field`.

```
class S256Point:
...
  @classmethod
  def parse(self, sec_bin):
    '''Возвращает объект типа Point из двоичных,
       а не шестнадцатеричных данных формата SEC'''
    if sec_bin[0] == 4:     ❶
      x = int.from_bytes(sec_bin[1:33], 'big')
      y = int.from_bytes(sec_bin[33:65], 'big')
      return S256Point(x=x, y=y)
    is_even = sec_bin[0] == 2     ❷
    x = S256Field(int.from_bytes(sec_bin[1:], 'big'))
    # правая часть уравнения y^2 = x^3 + 7
    alpha = x**3 + S256Field(B)
    # решить это уравнение для левой части
    beta = alpha.sqrt()     ❸
    if beta.num % 2 == 0:
      even_beta = beta
      odd_beta = S256Field(P - beta.num)
    else:
      even_beta = S256Field(P - beta.num)
      odd_beta = beta
    if is_even:
      return S256Point(x, even_beta)
    else:
      return S256Point(x, odd_beta)
```

❶ Несжатый формат SEC довольно прост.

❷ Четность координаты y задается в первом байте.

❸ Для получения координаты *y* в правой части уравнения, описывающего эллиптическую кривую, извлекается квадратный корень.

❹ Здесь определяется четность координаты *y* и возвращается правильно определенная точка.

Упражнение 2

Найдите открытый ключ в сжатом формате SEC, если имеются следующие секретные ключи.

- 5001
- 2019^5
- 0xdeadbeef54321

Подписи в формате DER

Помимо открытого ключа и точки на эллиптической кривой, необходимо подвергнуть сериализации подпись, представленную классом `Signature`. Как и в формате SEC, для этого придется закодировать две разные числовые величины — *r* и *s*. Но, к сожалению, в отличие от объекта типа `S256Point`, объект типа `Signature` нельзя сжать, поскольку величину *s* невозможно вывести непосредственно из величины *r*.

Для сериализации подписей (и многих других подобного рода данных) существует стандартный формат DER (Distinguished Encoding Rules — Особые правила кодирования). Именно этот формат и был использован Сатоши Накамото для сериализации подписей, главным образом потому, что он уже был определен в 2008 году и поддерживался в библиотеке OpenSSL, в то время применявшейся в биткойне, а кроме того, принять его было намного проще, чем разрабатывать новый стандарт.

Подпись определяется в формате DER следующим образом.

1. Сначала задается префиксный байт `0x30`.
2. Затем кодируется длина остальной подписи (обычно байтом `0x44` или `0x45`) и присоединяется к префиксному байту.
3. Далее присоединяется маркерный байт `0x02`.
4. После этого величина `r` кодируется в виде целого числа, представленного в обратном порядке следования байтов, но предваряется байтом `0x00`, если первый байт величины `r` меньше или равен `0x80`. К полученной в

итоге длине величины r присоединяется кодировка этой величины и все это вместе взятое добавляется к предыдущему результату.

5. Далее присоединяется маркерный байт 0x02.
6. И наконец, величина s кодируется в виде целого числа, представленного в обратном порядке следования байтов, но предваряется байтом 0x00, если первый байт величины s меньше или равен 0x80. К полученной в итоге длине величины s присоединяется кодировка этой величины, и все это вместе взятое добавляется к предыдущему результату.

Правила кодирования величин r и s, если их первый байт меньше или равен 0x80, установлены потому, что формат DER служит для общих целей кодирования и позволяет кодировать отрицательные числа. Если в первом бите числа установлена 1, то число отрицательное. Все числа в подписи, формируемой по алгоритму ECDSA, положительные, и поэтому их кодировки предваряются байтом 0x00, если в первом бите числа установлен 0, что равнозначно условию, когда первый байт меньше или равен 0x80.

Структура формата DER наглядно представлена на рис. 4.4.

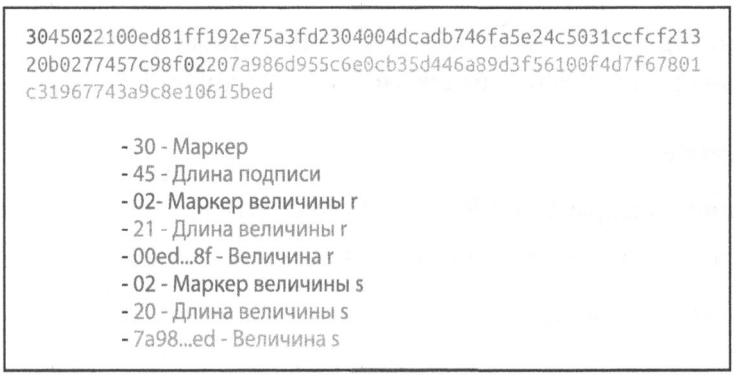

Рис. 4.4. Формат DER

Если известно, что r — 256-разрядное число, то величина r будет выражена максимум 32 байтами в обратном порядке. И вполне возможно, что первый ее байт может оказаться меньше или равным 0x80, поэтому по правилу из п. 4 приведенной выше процедуры величина r может быть выражена максимум 33 байтами. Но если r — относительно небольшое число, то оно может быть выражено меньшим, чем 32, количеством байтов. Это же справедливо для величины s и правила ее кодировки из п. 6 приведенной выше процедуры.

Ниже показано, каким образом программируется подпись в формате DER.

```
class Signature:
...
  def der(self):
  rbin = self.r.to_bytes(32, byteorder='big')
  # удалить все пустые байты в начале
  rbin = rbin.lstrip(b'\x00')
  # если в массиве rbin содержится единичный бит, добавить \x00
  if rbin[0] & 0x80:
    rbin = b'\x00' + rbin
  result = bytes([2, len(rbin)]) + rbin     ❶
  sbin = self.s.to_bytes(32, byteorder='big')
  # удалить все пустые байты в начале
  sbin = sbin.lstrip(b'\x00')
  # если в массиве sbin содержится единичный бит, добавить \x00
  if sbin[0] & 0x80:
   sbin = b'\x00' + sbin
  result += bytes([2, len(sbin)]) + sbin
  return bytes([0x30, len(result)]) + result
```

❶ Начиная с версии Python 3, появилась возможность преобразовать список чисел в их байтовые эквиваленты, сделав вызов `bytes([`*`целое_число1, целое_число2`*`])`.

В общем, кодировать величины `r` и `s` подобным способом неэффективно из-за наличия по меньшей мере шести байтов, которые не строго необходимы.

Упражнение 3

Представьте в формате DER подпись со следующими величинами *r* и *s*:

```
r = 0x37206a0610995c58074999cb9767b87af4c4978db68c06e8e6e81d282047a7c6

s = 0x8ca63759c1157ebeaec0d03cecca119fc9a75bf8e6d0fa65c841c8e2738cdaec
```

Кодировка Base58

На начальной стадии развития биткойнам присваивались открытые ключи, указанные в несжатом формате SEC, а затем они возвращались обратно с помощью подписей в формате DER. По причинам, объясняемым в главе 6, применение именно такого простого сценария оказалось нерациональным для хранения *вывода неизрасходованных транзакций* (Unspent Transaction Output — UTXO) и несколько менее безопасным, чем сценарии, чаще всего применяемые теперь. А до тех пор рассмотрим назначение адресов и порядок их кодирования.

Передача открытого ключа

Чтобы Алисе заплатить Бобу, ей нужно знать, куда отправить деньги. Это справедливо не только для биткойна, но и для любого другого способа оплаты. А поскольку биткойн является цифровым предъявительским финансовым инструментом, адрес может служить чем-то вроде открытого ключа в алгоритме криптографии с открытым ключом. Но, к сожалению, формат SEC (и особенно его несжатая форма) оказывается слишком длинным (65 или 33 байта). А кроме того, 65 или 33 байта исходно представлены в двоичной, а не в удобочитаемой форме.

Здесь необходимо принять во внимание три основных соображения. Во-первых, открытый ключ должен быть удобочитаемым, чтобы его можно было легко записать от руки или безошибочно сообщить, например, по телефону. Во-вторых, открытый ключ должен быть коротким, т.е. не настолько длинным, чтобы стать неудобным в обращении. И в-третьих, он должен быть безопасным, чтобы затруднить совершение ошибок.

Как в таком случае добиться удобочитаемости, сжатости и безопасности? Если выразить формат SEC в шестнадцатеричной форме (по 4 бита на каждый символ), то его длина удвоится (до 130 или 166 байтов). А можно ли достичь лучшего результата?

Можно, конечно, воспользоваться кодировкой Base64, в которой на каждый символ приходится 6 битов, а в итоге длина несжатого формата SEC составляет 87 символов, тогда как длина сжатого формата SEC — 44 символа. Но, к сожалению, кодировка Base64 подвержена ошибкам, поскольку многие буквы и числа выглядят в ней очень похожими (например, `0` и `O`, `l` и `I`, `-` и `_`). Если исключить такие символы, можно достичь результата с хорошей удобочитаемостью и приличной сжатостью (около 5,86 бита на каждый символ). И в конце можно добавить контрольную сумму, чтобы обеспечить простоту обнаружения ошибок.

Такая кодировка называется *Base58*. Вместо шестнадцатеричной формы (по основанию 16) или кодировки Base64 числа можно теперь представлять в кодировке Base58.

Конкретный механизм представления символов в кодировке Base58 следующий: употребляются все цифры, прописные и строчные буквы, кроме упомянутых выше символов `0`/`O` и `l`/`I`. В итоге остается 10 + 26 + 26 – 4 = 58 символов. Каждый из этих символов представляет отдельную цифру в кодировке

Base58. Ниже показано, как определяется функция для представления символов в кодировке Base58.

```
BASE58_ALPHABET =
      '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
...
def encode_base58(s):
  count = 0
  for c in s:     ❶
    if c == 0:
      count += 1
    else:
      break
    num = int.from_bytes(s, 'big')
    prefix = '1' * count
    result = ''
    while num > 0:     ❷
      num, mod = divmod(num, 58)
      result = BASE58_ALPHABET[mod] + result
  return prefix + result     ❸
```

❶ Этот цикл предназначен для определения количества нулевых начальных байтов. Их потребуется добавить снова в самом конце.

❷ В цикле определяется, какую именно цифру в кодировке Base58 следует употребить.

❸ И наконец, все подсчитанные ранее начальные нули присоединяются в начале получаемого результата, ведь иначе они не будут отображены как снабженные префиксом. Подобная неприятность происходит с оплатой по хешу открытого ключа (p2pkh; подробнее об этом читайте в главе 6).

Начиная с версии Python 3 приведенная выше функция способна получать любые байты и преобразовывать их в кодировку Base58.

Почему кодировка Base58 выходит из употребления

Кодировка Base58 применялась долгое время, и хотя она немного проще для сообщения адресов, чем, например, кодировка Base64, на самом деле она не очень удобна. Большинство пользователей предпочитают копировать и вставлять адреса, и если вам когда-нибудь приходилось сообщать адрес в кодировке Base58 голосом, то вам должно быть хорошо известно, какая это мука.

В этом отношении намного лучше кодировка по новому стандарту Bech32, определенному в протоколе BIP0173. В кодировке

Bech32 употребляется алфавит, состоящий из 32 символов, включая только цифры и строчные буквы, кроме 1, b, i и o. До сих пор эта кодировка применялась только для протокола Segwit (см. главу 13).

Упражнение 4

Преобразуйте приведенные ниже шестнадцатеричные значения в двоичную форму, а затем представьте их в кодировке Base58.

- `7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d`
- `eff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c`
- `c7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6`

Формат адреса

Длина 264 бита сжатого формата SEC слишком велика, не говоря уже о несколько пониженной безопасности (см. главу 6). Чтобы сократить адрес и повысить безопасность, можно воспользоваться хеш-функцией ripemd160.

Если пользоваться форматом SEC непосредственно, можно существенно сократить длину адреса с 33 до 20 байтов. Ниже поясняется, как образуется адрес биткойна.

1. Формирование адреса для сети mainnet начинается с префиксного байта `0x00`, а для сети testnet — с префиксного байта `0x6f`.
2. Затем выбирается формат SEC (сжатый или несжатый) и выполняется сначала хеш-функция sha256, а затем — хеш-функция ripemd160 (их сочетание называется хеш-функцией hash160).
3. Далее префикс, заданный в п. 1, объединяется с хешем, получаемым в п. 2.
4. После этого над результатом, получаемым в п. 3, выполняется хеш-функция hash256, и в конечном счете получаются первые четыре байта.
5. И наконец, результаты выполнения пп. 3 и 4 объединяются и преобразуются в кодировке Base58.

Результат выполнения п. 4 из описанного выше процесса называется *контрольной суммой*. Пункты 4 и 5 данного процесса можно выполнить сразу следующим образом:

```
def encode_base58_checksum(b):
  return encode_base58(b + hash256(b)[:4])

def hash160(s):
'''sha256 followed by ripemd160'''
return hashlib.new('ripemd160', hashlib.sha256(s).digest()).digest()    ❶
```

❶ Следует иметь в виду, что хеш-функция sha256 выполняется при вызове `hashlib.sha256(s).digest()`, в объемлющем его вызове — хеш-функция ripemd160.

Что такое testnet

testnet — это параллельная сеть биткойна, специально предназначенная для применения разработчиками. Монеты в этой сети ничего не стоят, а для того, чтобы относительно просто найти в ней блок, требуется подтверждение работы. Цепочка в сети mainnet на момент написания этой книги состояла из 550 000 блоков, тогда как в сети testnet их было значительно больше (около 1 450 000 блоков).

Кроме того, методы `hash160()` и `address()` можно обновить в классе `S256Point` следующим образом:

```
class S256Point:
...
  def hash160(self, compressed=True):
    return hash160(self.sec(compressed))

  def address(self, compressed=True, testnet=False):
    '''Возвращает адресную строку'''
    h160 = self.hash160(compressed)
    if testnet:
      prefix = b'\x6f'
    else:
      prefix = b'\x00'
    return encode_base58_checksum(prefix + h160)
```

Упражнение 5

Найдите адреса, соответствующие открытым ключам, которым отвечают следующие секретные ключи.

- 5002 (использовать несжатый формат SEC в сети testnet)
- 2020^5 (использовать сжатый формат SEC в сети testnet)
- 0x12345deadbeef (использовать сжатый формат SEC в сети mainnet)

Формат WIF

В рассматриваемом здесь случае открытый ключ является 256-разрядным числом. В общем, подвергать сериализации свой секретный ключ приходится нечасто, поскольку он не подлежит широковещательной передаче, ведь это была бы неудачная идея! Но иногда свой секретный ключ требуется передать из одного кошелька в другой, например из бумажного в программный кошелек.

По этой причине можно воспользоваться форматом WIF (Wallet Import Format — формат импорта кошелька). Этот формат предназначен для сериализации секретного ключа в удобочитаемом виде. В формате WIF используется та же самая кодировка Base58, что и для адресов.

Секретные ключи формируются в формате WIF следующим образом.

1. Формирование секретного ключа для сети mainnet начинается с префиксного байта `0x80`, а для сети `testnet` — с префиксного байта `0xef`.
2. Затем секретный ключ кодируется в виде 32 байтов, следующих в обратном порядке.
3. Если для формирования адреса открытого ключа использован сжатый формат SEC, то добавляется суффиксный байт `0x01`.
4. Далее префикс, заданный в п. 1, объединяется с секретным ключом, сериализированным в п. 2, а также с суффиксом, добавленным в п. 3.
5. После этого над результатом, полученным в п. 4, выполняется хеш-функция hash256, и в конечном счете получаются первые четыре байта.
6. И наконец, результаты выполнения пп. 4 и 5 объединяются и преобразуются в кодировке Base58.

Описанный выше процесс формирования секретного ключа в формате WIF можно теперь реализовать в методе `wif()`, определив его в классе `PrivateKey` следующим образом:

```
class PrivateKey
...
  def wif(self, compressed=True, testnet=False):
```

```
    secret_bytes = self.secret.to_bytes(32, 'big')
    if testnet:
      prefix = b'\xef'
    else:
      prefix = b'\x80'
    if compressed:
      suffix = b'\x01'
    else:
      suffix = b''
    return encode_base58_checksum(prefix + secret_bytes + suffix)
```

Упражнение 6

Сформируйте секретный ключ в формате WIF, исходя из следующих секретных данных.

- 5003 (сжатый формат SEC для сети testnet)
- 2021^5 (несжатый формат SEC для сети testnet)
- 0x54321deadbeef (сжатый формат SEC для сети mainnet)

Еще раз о прямом и обратном порядке следования байтов

Было бы весьма полезно знать, каким образом прямой и обратный порядок следования байтов реализуется в языке Python, поскольку в ряде последующих глав потребуется часто производить синтаксический анализ и сериализацию числовых данных как в обратном, так и в прямом порядке следования байтов. В частности, Сатоши Накомото часто пользовался прямым порядком следования байтов для биткойна, но, к сожалению, единого простого правила, устанавливающего, где следует использовать прямой, а где обратный порядок следования байтов, не существует. Напомним, что в формате SEC применяется обратный порядок следования байтов, как, впрочем, и в адресах формата WIF. Но начиная с главы 5 мы будем часто пользоваться кодировкой в прямом порядке следования байтов. Именно этому предмету и посвящены два следующих упражнения. А в последнем упражнении из этого раздела предлагается самостоятельно формировать адрес для сети testnet.

Упражнение 7

Напишите функцию `little_endian_to_int()`, получающую байты в том порядке, который принят в Python, интерпретирующую эти байты в прямом порядке следования и возвращающую целое число.

Упражнение 8

Напишите функцию `int_to_little_endian()`, выполняющую действия, противоположные функции из упражнения 7.

Упражнение 9

Сформируйте самостоятельно адрес для сети testnet, используя самые длинные секретные данные, которые вам только известны. Это очень важно, поскольку в сети testnet действует немало ботов, пытающихся украсть циркулирующие в ней монеты. Непременно запишите на чем-нибудь эти секретные данные! Они вам еще пригодятся в дальнейшем для подписания транзакций.

Заключение

В этой главе было показано, каким образом производится сериализация самых разных структур данных, созданных в предыдущих главах. А теперь перейдем к синтаксическому анализу и разъяснению понятия транзакций.

ГЛАВА 5

Транзакции

В основу биткойна положены транзакции. Проще говоря, транзакции переводят средства от одного субъекта экономической деятельности к другому. Как будет показано в главе 6, такими "субъектами" в данном случае выступают "умные контракты", но не будем забегать вперед. Выясним сначала, чем являются транзакции в биткойне, как они выглядят и как анализируются синтаксически.

Составляющие транзакции

В общем, транзакция образуется всего лишь из четырех составляющих. Эти составляющие таковы.

1. Версия.
2. Вводы.
3. Выводы.
4. Время блокировки.

Полезно иметь общее представление об этих составляющих. В частности, версия обозначает, какие дополнительные средства применяются в транзакции, вводы определяют затрачиваемые биткойны, выводы — куда биткойны направляются, а время блокировки — момент, с которого данная транзакция становится действительной. Все эти составляющие транзакции будут подробнее рассмотрены в дальнейшем.

На рис. 5.1 приведен шестнадцатеричный дамп (т.е. вывод из оперативной памяти) типичной транзакции, где разными оттенками серого обозначены отдельные ее составляющие.

С учетом вышесказанного, мы можем построить класс транзакции `Tx`, как показано ниже.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.1. Составляющие транзакции: версия, вводы, выводы, время блокировки

```
class Tx:
  def __init__(self, version, tx_ins, tx_outs, locktime,
               testnet=False):
    self.version = version     ❶
    self.tx_ins = tx_ins
    self.tx_outs = tx_outs
    self.locktime = locktime
    self.testnet = testnet     ❷

  def __repr__(self):
    tx_ins = ''
    for tx_in in self.tx_ins:
      tx_ins += tx_in.__repr__() + '\n'
      tx_outs = ''
    for tx_out in self.tx_outs:
      tx_outs += tx_out.__repr__() + '\n'
    return 'tx: {}\nversion: {}\ntx_ins:\n{}tx_outs:\n{}locktime: {}'
           .format(
                   self.id(),
                   self.version,
                   tx_ins,
                   tx_outs,
                   self.locktime,
            )

  def id(self):     ❸
    '''Хеш транзакции в удобочитаемой шестнадцатеричной форме'''
    return self.hash().hex()

  def hash(self):     ❹
    '''Хеш устаревшей сериализации в двоичной форме'''
    return hash256(self.serialize())[::-1]
```

❶ Вводы и выводы — слишком общие понятия, поэтому здесь определяются конкретные вводы транзакции. А конкретные типы объектов будут определены в дальнейшем.

❷ Чтобы проверить данную транзакцию на достоверность, необходимо знать, в какой сети она выполняется.

❸ Для поиска транзакций обозреватели пользуются методом `id()`. Он реализует хеш-функцию hash256 над транзакцией в шестнадцатеричной форме.

❹ Метод `hash()` реализует хеш-функцию hash256, выполняемую над результатом сериализации в прямом порядке следования байтов. Следует, однако, иметь в виду, что метод `serialize()` пока еще не определен, и до тех пор метод `hash()` не сможет действовать нормально.

Остальная часть этой главы посвящена синтаксическому анализу транзакций. А пока что мы можем определить в классе `Tx` следующий метод для синтаксического анализа транзакций:

```
class Tx:
  ...

  @classmethod    ❶
  def parse(cls, serialization):
    version = serialization[0:4]    ❷
    ...
```

❶ Это должен быть метод класса, поскольку в результате сериализации будет возвращен новый экземпляр объекта типа `Tx`.

❷ Здесь предполагается, что переменной `serialization` присваивается массив байтов.

Этого могло бы оказаться достаточно, но ведь транзакция может быть очень крупной. В идеальном случае ее синтаксический анализ желательно произвести из *потока*. Ведь это позволит не затребовать сериализированную транзакцию полностью, прежде чем приступить к ее синтаксическому анализу, завершив его как можно раньше при неудачном исходе, а следовательно, действовать более эффективно. Таким образом, исходный код для синтаксического анализа транзакции будет выглядеть как

```
class Tx:
  ...
  @classmethod
  def parse(cls, stream):
    serialized_version = stream.read(4)    ❶
    ...
```

❶ Метод `read()` позволит оперативно производить синтаксический анализ транзакции, не ожидая ввода-вывода.

С технической точки зрения такой подход оказывается более предпочтительным, поскольку в качестве потока может служить сокетное соединение в сети или дескриптор файла. Синтаксический анализ содержимого потока можно начать сразу, а не ждать завершения передачи всех данных или предварительного их чтения. Определенный выше метод `parse()` будет в состоянии обрабатывать любой поток и возвращать именно тот объект типа `Tx`, который нам требуется.

Версия

Если отображается номер версии какого-нибудь программного компонента (как, например, на рис. 5.2), этот номер предназначается для предоставления получателю сведений о привязке данного компонента к указанной версии. Так, если вы работаете в операционной системе Windows 3.1, номер ее версии существенно отличается от версии Windows 8 или Windows 10. Вы можете, конечно, обойтись и названием "Windows" операционной системы, но если вы упомянете и номер ее версии, он поможет вам вспомнить или выяснить ее функциональные возможности и интерфейсы API, используемые для программирования.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.2. Версия транзакции

Аналогично у транзакций биткойна имеются свои номера версий. Как правило, транзакция биткойна имеет номер версии, равный 1, но иногда он может быть равным 2. В частности, для тех транзакций, в которых применяется код операции `OP_CHECKSEQUENCEVERIFY` по протоколу BIP0112, требуется номер версии больше 1.

Обратите на рис. 5.2 внимание на то, что номер версии транзакции обозначается в шестнадцатеричной форме как `01000000`, которая не очень похожа на номер версии 1. Но в шестнадцатеричной форме номер 1 фактически интерпретируется как целое число в прямом порядке следования байтов, как обсуждалось в главе 4.

Упражнение 1

Напишите новую версию той части метода `parse()`, где производится синтаксический анализ транзакции. Чтобы сделать это надлежащим образом, вам придется преобразовать 4 байта в целое число, представленное байтами в прямом порядке их следования.

Вводы

Каждый ввод указывает на вывод предыдущей транзакции (рис. 5.3). Это обстоятельство требует дополнительного разъяснения, если оно не вполне очевидно поначалу.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.3. Вводы транзакции

На вводах транзакции биткойна расходуются выводы предыдущей транзакции. Это означает, что биткойны должны быть получены, прежде чем они будут потрачены. Вводы транзакции ссылаются на биткойны, которые принадлежат их владельцу. Для каждого ввода требуются следующие составляющие.

- Ссылка на полученные ранее биткойны.
- Подтверждение, что они принадлежат своему владельцу, который имеет право их потратить.

Во второй составляющей применяется алгоритм ECDSA (см. главу 3). Вам как владельцу биткойнов вряд ли захочется, чтобы подтверждение их принадлежности лично вам было кем-то подделано, и поэтому большинство вводов транзакций содержат подписи, которые могут сделать только владельцы секретных ключей.

Поле вводов транзакции может содержать не один ввод. Это можно сравнить с употреблением одного счета для оплаты трапезы стоимостью 70 долларов США или двух счетов на 50 и 20 долларов США. В первом случае требуется лишь один ввод (в данном случае — счет), а во втором случае — два ввода. Но иногда вводов может быть и больше. По аналогии трапезу

стоимостью 70 долларов США можно оплатить 14 счетами по 5 долларов США или даже 7000 счетами по одному центу. Это равнозначно 14 или 7000 вводам транзакции.

Количество вводов указывается после версии транзакции. Так, на рис. 5.4 оно выделено светло-серым и равно 01.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b48304502100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.4. Количество вводов транзакции

Как видите, количество вводов обозначено байтом 01, а следовательно, у данной транзакции лишь один ввод. И здесь может возникнуть искушение допустить, что количество вводов транзакции всегда обозначается одним байтом, но на самом деле это не так. Ведь один байт состоит из 8 битов, а следовательно, одним байтом нельзя выразить количество вводов свыше 255.

И здесь вступают в действие *варианты*. Сокращенно вариант обозначает *переменное целое* (variable integer), т.е. способ кодирования целого числа количеством байтов в пределах от 0 до $2^{64} - 1$. Для обозначения количества вводов транзакции можно было бы, конечно, всегда резервировать 8 байтов, но это было бы напрасной тратой свободного места, если предполагается относительно небольшое количество вводов (например, меньше 20), что, как правило, характерно для обычной транзакции. А варианты помогают сэкономить свободное место. Принцип их действия поясняется в приведенном ниже разделе.

Варианты

Варианты как переменные целые числа действуют по следующим правилам.

- Если число меньше 253, оно кодируется одним байтом (например, 100 → 0x64).
- Если число находится в пределах от 253 до $2^{64} - 1$, то сначала кодируется число 253 одним байтом (fd), а затем заданное число — 2 байтами в прямом порядке их следования (например, 255 → 0xfdff00, 555 → 0xfd2b02).

- Если число находится в пределах от 2^{16} до $2^{32} - 1$, то сначала кодируется число 254 одним байтом (fe), а затем заданное число — 4 байтами в прямом порядке их следования (например, 70015 → 0xfe7f110100).
- Если число находится в пределах от 2^{32} до $2^{64} - 1$, то сначала кодируется число 255 одним байтом (ff), а затем заданное число — 8 байтами в прямом порядке их следования (например, 18005558675309 → 0xff6dc7ed3e60100000).

Для синтаксического анализа и сериализации полей вариантов можно воспользоваться следующими двумя функциями из исходного файла helper.py:

```
def read_varint(s):
  '''Эта функция читает переменное целое число из потока'''
  i = s.read(1)[0]
  if i == 0xfd:
    # 0xfd означает следующие два байта, обозначающие число
    return little_endian_to_int(s.read(2))
  elif i == 0xfe:
    # 0xfe означает следующие четыре байта, обозначающие число
    return little_endian_to_int(s.read(4))
  elif i == 0xff:
    # 0xff означает следующие восемь байтов, обозначающих число
    return little_endian_to_int(s.read(8))
  else:
    # все остальное является всего лишь целым числом
  return i

def encode_varint(i):
  '''Кодирует целое число в виде варианта'''
  if i < 0xfd:
    return bytes([i])
  elif i < 0x10000:
    return b'\xfd' + int_to_little_endian(i, 2)
  elif i < 0x100000000:
    return b'\xfe' + int_to_little_endian(i, 4)
  elif i < 0x10000000000000000:
    return b'\xff' + int_to_little_endian(i, 8)
  else:
    raise ValueError('integer too large: {}'.format(i))
```

Функция read_varint() читает из потока вариант в закодированном виде и возвращает целое число. А функция encode_varint() делает совершенно

обратное, т.е. принимает целое число и возвращает вариант в байтовом представлении.

Каждый ввод состоит из четырех полей. Два первых поля содержат ссылки на вывод предыдущей транзакции, а в двух последних полях определяется порядок расходования вывода предыдущей транзакции. Ниже приведено содержимое этих полей.

- Идентификатор предыдущей транзакции.
- Индекс предыдущей транзакции.
- Сценарий ScriptSig.
- Последовательность.

Как пояснялось выше, каждый ввод содержит ссылку на вывод предыдущей транзакции. Идентификатором предыдущей транзакции служит результат выполнения хеш-функции hash256 над содержимым предыдущей транзакции. Этим предыдущая транзакция определяется однозначно, поскольку вероятность коллизии хеш-функций в данном случае невелика.

Как станет ясно в дальнейшем, у каждой транзакции должен быть хотя бы один вывод, хотя их может быть и много. Поэтому необходимо точно определить, какой именно вывод расходуется в *самой* транзакции, что зафиксировано в индексе предыдущей транзакции. Следует, однако, иметь в виду, что идентификатор предыдущей транзакции представлен 32 байтами, а ее индекс — 4 байтами, причем в прямом порядке их следования.

Сценарий ScriptSig имеет отношение к языку Script для написания умных контрактов в биткойне, подробно рассматриваемому в главе 6. А до тех пор сценарий ScriptSig можно рассматривать как средство для открытия запертого на ключ денежного ящика, что может сделать только владелец вывода транзакции. Поле ScriptSig имеет переменную, а не фиксированную длину, как большинство рассмотренных до сих пор полей. И поскольку поле переменной длины требует точного определения его длины, оно предваряется вариантом, в котором указывается его длина.

Последовательность первоначально задумывалась как способ обращения с полем блокировки времени для совершения того, что Сатоши Накамото назвал "высокочастотными сделками" (см. приведенную далее врезку "Последовательность и время блокировки"). Но в настоящее время применяются способ RBF (Replace-By-Fee — Замена транзакции с повышением платы за нее) и

операция OP_CHECKSEQUENCEVERIFY. Последовательность также представлена в прямом порядке следования байтов и занимает 4 байта.

Описанные выше поля ввода транзакции выделены разными оттенками серого на рис. 5.5.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d1 0000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278feffffff 02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.5. Поля ввода транзакции: идентификатор и индекс предыдущей транзакции, сценарий ScriptSig и последовательность

Последовательность и время блокировки

Первоначально Сатоши Накамото требовалось, чтобы поля последовательности и времени блокировки использовались для совершения так называемых "высокочастотных сделок". По существу, он предусматривал способ взаимной оплаты торгующими сторонами без совершения большого числа транзакции в цепочке блоков. Так, если Алиса платит Бобу за что-нибудь одно x биткойнов, а Боб платит Алисе за что-нибудь другое y биткойнов (например, если $x > y$), то Алисе достаточно заплатить $x - y$ биткойнов вместо того, чтобы совершать две отдельные транзакции в цепочке блоков. То же самое можно было бы сделать и в том случае, если бы Алиса и Боб совершили 100 взаимных транзакций, а следовательно, такой способ позволяет свести целый ряд транзакций к единственной транзакции.

Замысел Сатоши Накамото состоял в том, чтобы непрерывно обновлять мини-журнал учета операций между торгующими сторонами, устанавливаемый в цепочке блоков. Сатоши Накамото намеревался использовать поля последовательности и времени блокировки для обновления транзакции при высокочастотной сделке всякий раз, когда между торгующими сторонами происходила новая оплата. У транзакции торговой сделки должно было быть два ввода (один — от Алисы, а другой — от Боба) и два вывода (один — к Алисе, а другой — к Бобу). Транзакция торговой сделки должна была начинаться с нулевой последовательностью и отдаленным временем блокировки (например, через 500 блоков, чтобы она была действительна в течение 500 блоков). Это должна была быть основная транзакция, где у Алисы и Боба были бы одинаковые вложенные ими суммы.

После первой транзакции, где Алиса платит Бобу *x* биткойнов, последовательность каждого ввода должна стать равной 1. А после второй транзакции, где Боб платит Алисе *y* биткойнов, она должна стать равной 2. Подобным способом можно было бы свести многие оплаты к единственной транзакции в цепочке блоков, при условии, что они произошли до истечения времени блокировки.

Но, к сожалению, такой способ, как бы ловко он ни был придуман, позволяет добытчику криптовалюты очень легко совершить подлог. В рассматриваемом здесь примере таким добытчиком криптовалюты мог бы быть Боб, которому ничего не стоило бы просто проигнорировать обновленную транзакцию торговой сделки с номером последовательности 2 и добыть транзакцию торговой сделки с номером последовательности 1, тем самым обманув Алису на *y* биткойнов. В дальнейшем был придуман более совершенный способ с так называемыми "платежными каналами" в качестве основания для протокола Lightning Network (Молниеносная сеть).

Итак, зная, чем являются поля вводов транзакции, можно теперь приступить к созданию класса `TxIn` на языке Python следующим образом:

```
class TxIn:
  def __init__(self, prev_tx, prev_index, script_sig=None,
               sequence=0xffffffff):
    self.prev_tx = prev_tx
    self.prev_index = prev_index
    if script_sig is None:    ❶
      self.script_sig = Script()
    else:
      self.script_sig = script_sig
      self.sequence = sequence

  def __repr__(self):
    return '{}:{}'.format(
      self.prev_tx.hex(),
      self.prev_index,
    )
```

❶ Стандартно поле ScriptSig пустое.

Приведенный выше класс требует некоторых пояснений. Во-первых, в нем не задана сумма для каждого ввода, поскольку затраты неизвестны до тех пор, пока в блокчейне (т.е. в цепочке блоков) не будут обнаружены израсходованные транзакции. И во-вторых, неизвестно даже, отпирает ли, так

сказать, транзакция нужный денежный ящик, если ничего неизвестно о предыдущей транзакции. В каждом узле должно проверяться, отпирает ли данная транзакция нужный денежный ящик и не тратятся ли при этом несуществующие биткойны. О том, как это делается, речь пойдет в главе 7.

Синтаксический анализ сценариев

Более подробно о синтаксическом анализе сценариев на языке Script речь пойдет в главе 7, а пока преобразуем объект типа `Script` из шестнадцатеричной формы в форму Python, как показано ниже.

```
>>> from io import BytesIO
>>> from script import Script     ❶
>>> script_hex = ('6b483045022100ed81ff192e75a3fd2304004dcadb\
746fa5e24c5031ccf cf21320b0277457c98f02207a986d955c6e0cb35d446a\
89d3f56100f4d7f67801c31967743a9c8\e10615bed01210349fc4e631e3624a\
545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278\a')
>>> stream = BytesIO(bytes.fromhex(script_hex))
>>> script_sig = Script.parse(stream)
>>> print(script_sig)
3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b\
0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801c\
31967743a9c8e10615bed01 0349fc4e631e3624a545de3f89f5d8684c7b\
8138bd94bdd531d2e213bf016b278a
```

❶ Более подробно класс `Script` рассматривается в главе 6, а сейчас просто убедитесь, что требуемый объект создан в методе `Script.parse()`.

Упражнение 2

Напишите ту часть метода `parse()` из класса `Tx` и метода `parse()` из класса `TxIn`, которая отвечает за синтаксический анализ вводов транзакции.

Выводы

Как упоминалось в предыдущем разделе, вывод транзакции определяет, куда именно направляются биткойны. У каждой транзакции должно быть от одного и больше выводов. А зачем вообще иметь больше одного вывода? Ведь для обмена можно, например, обойтись и формированием пакета транзакций, чтобы заплатить сразу многим лицам вместо того, чтобы формировать по одной транзакции для каждого лица, требующего биткойны к оплате.

Сериализация выводов, как и вводов, начинается с указания количества выводов в виде варианта. Так, на рис. 5.6 это количество выделено светло-серым и равно 02.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff 02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.6. Количество выводов транзакции

Каждый вывод транзакции состоит из двух полей: суммы и сценария ScriptPubKey. Сумма обозначает количество присвоенных биткойнов и указывается в единицах, называемых "сатоши" и составляющих 1/100000000 долю биткойна. Это позволяет очень точно делить биткойны вплоть до 1/300-й доли цента США (на момент написания данной книги). Абсолютный максимум данной суммы составляет асимптотический предел, равный 21 миллиону биткойнов или 2100 триллионам Сатоши. Эта сумма оказывается больше 2^{32} (т.е. около 4,3 миллиарда), а следовательно, она хранится в 64 битах или 8 байтах, которые сериализуются в прямом порядке их следования.

Сценарий ScriptPubKey, как и сценарий ScriptSig, имеет отношение к языку Script, используемому для написания умных контрактов в биткойне. Данный сценарий можно рассматривать как запертый на ключ денежный ящик, который может открыть только владелец ключа от этого ящика. Это можно сравнить с односторонним сейфом, который может принимать вклады от кого угодно, но открыть его может только владелец данного сейфа. Более подробно сценарий ScriptPubKey рассматривается в главе 6. Поле сценария ScriptPubKey, как и поле сценария ScriptSig, имеет переменную длину, конкретное значение которой указывается в варианте перед самим полем.

Описанные выше поля вывода транзакции выделены разными оттенками серого на рис. 5.7.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.7. Поля вывода транзакции, в которых отображаются доступная сумма и сценарий ScriptPubKey (последний — по индексу 0)

Множество UTXO

Сокращение UTXO обозначает *вывод неизрасходованных транзакций*. Вся совокупность выводов неизрасходованных транзакции в любой данный момент называется *множеством UTXO*. Такие множества важны потому, что в любой момент времени представляют все доступные для расходования или находящиеся в обращении биткойны. Множество UTXO должно отслеживаться в полных узлах сети, а его индексирование ускоряет проверку достоверности новых транзакций.

Например, соблюсти правило, запрещающее двойное расходование, нетрудно, найдя вывод предыдущей транзакции в множестве UTXO. Так, если на вводе новой транзакции используется вывод транзакции, отсутствующий в множестве UTXO, это означает попытку двойного расходования или употребления несуществующего вывода, а следовательно, такая операция считается недействительной. Иметь в своем распоряжении множество UTXO удобно и для проверки достоверности транзакций. Как будет показано в главе 6, чтобы проверить транзакцию на достоверность, необходимо найти сумму и сценарий ScriptPubKey из вывода предыдущей транзакции. Поэтому, имея в своем распоряжении множество UTXO, можно значительно ускорить проверку транзакций на достоверность.

Теперь можно приступить к определению класса `TxOut`, представляющего выводы транзакции, как показано ниже.

```
class TxOut:

  def __init__(self, amount, script_pubkey):
    self.amount = amount
    self.script_pubkey = script_pubkey

  def __repr__(self):
    return '{}:{}'.format(self.amount, self.script_pubkey)
```

Упражнение 3

Напишите ту часть метода `parse()` из класса `Tx` и метода `parse()` из класса `TxOut`, которая отвечает за синтаксический анализ выводов транзакции.

Время блокировки

Время блокировки — это способ отложить транзакцию на время. Так, транзакция со временем блокировки 600000 не может войти в блокчейн до тех пор, пока не будет достигнут блок 600001. Первоначально время блокировки интерпретировалось как способ совершения высокочастотных сделок (см. раздел "Последовательность и время блокировки", приведенную ранее в этой главе), что оказалось небезопасно. Если время блокировки больше или равно 500000000, то оно служит в качестве отметки времени в операционной системе Unix. А если время блокировки меньше 500000000, то оно обозначает номер блока. Таким образом, транзакции могут быть подписаны, но не израсходованы до тех пор, пока не настанет определенный момент времени в системе Unix или не будет достигнута определенная высота блока в цепочке блоков.

Когда время блокировки игнорируется

Время блокировки игнорируется, если номера последовательностей для каждого ввода равны `ffffffff`.

Сериализация времени блокировки осуществляется в прямом порядке следования 4 байтов, как выделено серым на рис. 5.8.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 5.8. Время блокировки

Главная трудность в применении времени блокировки состоит в том, что у получателя транзакции нет никакой уверенности, что она окажется пригодной, когда придет время блокировки. Это можно сравнить с банковским чеком, который датирован задним числом и от оплаты которого можно отказаться. Отправитель транзакции со временем блокировки может израсходовать ее вводы еще до того, как она войдет в блокчейн, в результате чего такая транзакция станет недействительной во время блокировки.

До появления протокола BIP0065 время блокировки применялось ограниченно. А в протоколе BIP 0065 была введена операция `OP_CHECKLOCKTIMEVERIFY`, благодаря которой время блокировки стало

приносить большую пользу, делая вывод неизрасходованным до тех пор, пока не наступит определенное время блокировки.

Упражнение 4

Напишите ту часть метода `parse()` из класса `Tx`, которая отвечает за синтаксический анализ времени блокировки.

Упражнение 5

Что содержит поле ScriptSig на втором вводе, поле ScriptPubKey — на первом выводе и поле суммы — на втором выводе приведенной ниже транзакции?

```
010000000456919960ac691763688d3d3bcea9ad6ecaf875df5339e148\
a1fc61c6ed7a069e0100
00006a47304402204585bcdef85e6b1c6af5c2669d4830ff86e42dd205\
c0e089bc2a821657e951
c002201024a10366077f87d6bce1f7100ad8cfa8a064b39d4e8fe4ea13\
a7b71aa8180f012102f0
da57e85eec2934a82a585ea337ce2f4998b50ae699dd79f5880e253daf\
afb7feffffffeb8f51f4
038dc17e6313cf831d4f02281c2a468bde0fafd37f1bf882729e7fd\
3000000006a473044022078
99531a52d59a6de200179928ca900254a36b8dff8bb75f5f5d71b1cd\
c26125022008b422690b84
61cb52c3cc30330b23d574351872b7c361e9aae3649071c1a7160121035\
d5c93d9ac96881f19ba
1f686f15f009ded7c62efe85a872e6a19b43c15a2937feffffff567bf\
40595119d1bb8a3037c35
6efd56170b64cbcc160fb028fa10704b45d775000000006a47304402204c7\
c7818424c7f7911da
6cddc59655a70af1cb5eaf17c69dadbfc74ffa0b662f02207599e08bc\
8023693ad4e9527dc42c3
4210f7a7d1d1ddfc8492b654a11e7620a0012102158b46fbdff65d0172
\b7989aec8850aa0dae49
abfb84c81ae6e5b251a58ace5cfeffffffd63a5e6c16e620f86f375925\
b21cabaf736c779f88fd
04dcad51d26690f7f345010000006a47304402200633ea0d3314bea0d95\
b3cd8dadb2ef79ea833
1ffe1e61f762c0f6daea0fabde022029f23b3e9c30f080446150b2385202\
8751635dcee2be669c
2a1686a4b5edf304012103ffd6f4a67e94aba353a00882e563ff2722eb4cf\
f0ad6006e86ee20df
e7520d55feffffff0251430f00000000001976a914ab0c0b2e98b1ab6dbf\
67d4750b0a56244948
a87988ac005a6202000000001976a9143c82d7df364eb6c75be8c80df2\
b3eda8db57397088ac46430600
```

Программная реализация транзакций

Итак, мы синтаксически проанализировали транзакцию. А теперь нужно сделать противоположное: сериализировать ее. Для этого начнем с класса `TxOut`:

```
class TxOut:
...
  def serialize(self):    ❶
    '''Возвращает результат сериализации вывода транзакции
       в последовательность байтов'''
    result = int_to_little_endian(self.amount, 8)
    result += self.script_pubkey.serialize()
    return result
```

❶ В этом методе предполагается сериализировать объект типа `TxOut` в последовательность байтов.

А теперь перейдем к классу `TxIn`:

```
class TxIn:
...
  def serialize(self):
    '''Возвращает результат сериализации ввода транзакции
       в последовательность байтов'''
    result = self.prev_tx[::-1]
    result += int_to_little_endian(self.prev_index, 4)
    result += self.script_sig.serialize()
    result += int_to_little_endian(self.sequence, 4)
    return result
```

И наконец, сериализируем транзакцию в классе `Tx`:

```
class Tx:
...
  def serialize(self):
    '''Возвращает результат сериализации транзакции
       в последовательность байтов'''
    result = int_to_little_endian(self.version, 4)
    result += encode_varint(len(self.tx_ins))
    for tx_in in self.tx_ins:
      result += tx_in.serialize()
    result += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
      result += tx_out.serialize()
    result += int_to_little_endian(self.locktime, 4)
    return result
```

Для сериализации объекта типа `Tx` в классах `TxIn` и `TxOut` использован метод `serialize()`. Следует, однако, иметь в виду, ни в одном из них плата за транзакцию не указана явно! Дело в том, что сумма платы за транзакцию подразумевается, как поясняется в следующем разделе.

Плата за транзакцию

Одно из согласованных в биткойне правил гласит: для любой немонетизирующей транзакции (подробнее о монетизирующих транзакциях речь пойдет в главе 9) сумма вводов должна быть больше или равна сумме выводов. А почему нельзя просто приравнять вводы и выводы транзакции? Дело в том, что если бы каждая транзакция ничего не стоила, то у добытчиков криптовалюты не было бы побудительных причин включать свои транзакции в блоки (см. главу 9). Именно плата за транзакции и побуждает добытчиков криптовалюты включать свои транзакции в блоки. Транзакции, не входящие в блоки (так называемые *транзакции из пула памяти*), не являются частью блокчейна, а следовательно, считаются незавершенными.

Плата за транзакцию — это просто разность сумм ее вводов и выводов. Именно эту разность добытчики криптовалюты и стремятся оставить себе. Если у вводов отсутствует поле суммы, ее придется найти, а для этого потребуется доступ к блокчейну вообще и к множеству UTXO в частности. Но сделать это нелегко, если не вести самостоятельно полный узел, ведь иначе придется довериться другому субъекту, чтобы получить от него требующуюся информацию.

Чтобы извлекать сведения о транзакциях, создадим новый класс `TxFetcher`, как показано ниже.

```
class TxFetcher:
  cache = {}

  @classmethod
  def get_url(cls, testnet=False):
    if testnet:
      return 'http://testnet.programmingbitcoin.com'
    else:
      return 'http://mainnet.programmingbitcoin.com'

  @classmethod
  def fetch(cls, tx_id, testnet=False, fresh=False):
    if fresh or (tx_id not in cls.cache):
      url = '{}/tx/{}.hex'.format(cls.get_url(testnet), tx_id)
      response = requests.get(url)
```

```
    try:
      raw = bytes.fromhex(response.text.strip())
    except ValueError:
      raise ValueError('unexpected response: {}'
           .format(response.text))
    if raw[4] == 0:
      raw = raw[:4] + raw[6:]
      tx = Tx.parse(BytesIO(raw), testnet=testnet)
      tx.locktime = little_endian_to_int(raw[-4:])
    else:
      tx = Tx.parse(BytesIO(raw), testnet=testnet)
    if tx.id() != tx_id:    ❶
      raise ValueError('not the same id: {} vs {}'
           .format(tx.id(), tx_id))
    cls.cache[tx_id] = tx
  cls.cache[tx_id].testnet = testnet
  return cls.cache[tx_id]
```

❶ Здесь проверяется, является ли идентификатор транзакции предполагаемым.

А почему бы не получить конкретный вывод транзакции вместо того, чтобы получать всю транзакцию в целом? Дело в том, что нам не следует полностью доверять третьей стороне! Получив всю транзакцию в целом, мы можем проверить ее идентификатор (результат выполнения хеш-функции hash256 над ее содержимым) и удостовериться, что мы получили именно ту транзакцию, которая нам и требовалась. А сделать это невозможно, не получив всю транзакцию в целом.

Почему не следует особенно доверять третьей стороне

В своем оригинальном очерке "Trusted Third Parties are Security Holes" (Доверенные третьи стороны: бреши в защите; `https://nakamotoinstitute.org/trusted-third-parties/`) Ник Сабо (Nick Szabo) красноречиво объяснил, что доверие третьим сторонам в получении от них правильных данных *нельзя* считать надлежащей нормой обеспечения безопасности. Ведь третья сторона может вести себя хорошо теперь, но вы не можете знать заранее, когда она подвергнется взлому, ее сотрудники станут самовольничать или вводить правила, противоречащие вашим интересам. Отчасти безопасность биткойна объясняется тем, что полученные данные всегда проверяются, а *не* принимаются на веру.

А теперь можно создать в классе TxIn метод для извлечения предыдущей транзакции, а также методы для получения суммы и сценария ScriptPubKey (он будет использоваться в главе 6) из вывода предыдущей транзакции, как показано ниже.

```
class TxIn:
...
  def fetch_tx(self, testnet=False):
    return TxFetcher.fetch(self.prev_tx.hex(), testnet=testnet)

  def value(self, testnet=False):
    '''Получает сумму из вывода, просматривая хеш транзакции.
       Возвращает сумму в сатоши'''
    tx = self.fetch_tx(testnet=testnet)
    return tx.tx_outs[self.prev_index].amount

  def script_pubkey(self, testnet=False):
    '''Получает сценарий ScriptPubKey просматривая хеш транзакции.
       Возвращает объект типа Script'''
    tx = self.fetch_tx(testnet=testnet)
    return tx.tx_outs[self.prev_index].script_pubkey
```

Расчет платы за транзакцию

Теперь, когда в классе TxIn имеется метод value(), позволяющий оценить количество биткойнов на каждом вводе, можно рассчитать плату за транзакцию.

Упражнение 6

Напишите метод fee() для класса Tx, чтобы рассчитать плату за транзакцию.

Заключение

В этой главе подробно описано, каким образом осуществляются синтаксический анализ и сериализация транзакций и каково назначение отдельных ее полей. Два из этих полей требуют дополнительного разъяснения, поскольку они непосредственно связаны с языком Script — языком составления умных контрактов в биткойне. Именно этому предмету и посвящена глава 6.

ГЛАВА 6

Язык Script

Возможность блокировать и разблокировать монеты является именно тем механизмом, который передает биткойн. *Блокировка* означает ссуду определенного количества биткойнов какому-то субъекту, а *разблокировка* — расходование некоторого количества полученных биткойнов.

В этой главе рассматриваются механизмы блокировки и разблокировки, которые зачастую называются *умным контрактом*. Криптография по эллиптическим кривым (см. главу 3) применяется в языке Script для проверки правильности авторизации транзакции (см. главу 5). По существу, язык Script позволяет людям убедиться, что они имеют право потратить определенное количество выводов неизрасходованных транзакций (UTXO). И хотя мы забегаем немного вперед, начнем все же с внутреннего механизма Script.

Внутренний механизм Script

Если вас смущает понятие умного контракта, не отчаивайтесь. "Умный контракт" — это просто изощренный способ назвать программируемый контракт, а "язык составления умных контрактов" — обыкновенный язык программирования. В биткойне для составления умных контрактов применяется язык программирования Script, на котором можно выразить условия для расходования биткойнов.

Биткойн является цифровым эквивалентом контракта в Script, а Script — языком со стековой организацией подобно языку Forth. В языке Script намеренно наложены определенные ограничения, чтобы исключить из него некоторые языковые средства. В частности, любые средства для организации циклов в Script отсутствуют, и потому этот язык не является полным по Тьюрингу.

Почему биткойн не является полным по Тьюрингу

Полнота по Тьюрингу в языке программирования, по существу, означает, что программа может выполняться циклически. Циклы являются полезной конструкцией в программировании, и поэтому вас может удивить, почему в языке Script отсутствует возможность организовать цикл.

Для этого имеется немало причин, но начнем их объяснение с выполнения программы. Всякий может создать программу на языке Script, которая будет выполнена в каждом полном узле сети. Если бы язык Script был полным по Тьюрингу, то вполне возможно, что цикл выполнялся бы бесконечно, а в итоге проверяющие узлы вошли бы в этот цикл, так и не выйдя из него. Подобным способом можно легко организовать атаку типа отказа в обслуживании (DoS) на сеть. Единственной программы с бесконечным циклом оказалось бы достаточно, чтобы сломать биткойн! И это была бы крупная систематическая уязвимость, защита от которой стала одной из главных причин, по которым в биткойне вообще и Script в частности исключена полнота по Тьюрингу. На платформе Ethereum, где применяется полный по Тьюрингу язык составления умных контрактов Solidity, проблема зацикливания решается оригинальным способом: контракты вынуждают расплачиваться за исполнение программы так называемым "газом". Бесконечный цикл будет испускать любой газ, который присутствует в контракте, поскольку он по определению будет выполняться бесконечное число раз.

Еще одна причина избежать полноты по Тьюрингу объясняется тем, что полные по Тьюрингу умные контракты с трудом поддаются анализу. Условия для выполнения контракта с полнотой по Тьюрингу очень трудно перечислить, а следовательно, нетрудно довести до непреднамеренного поведения, вызывающего программные ошибки. Такие ошибки в умном контракте означают, что монеты уязвимы к непреднамеренной трате, а значит, участники такого контракта могут потерять свои деньги. И такие ошибки носят не только теоретический характер. Они стали причиной главного затруднения в DAO (Decentralized Autonomous Organization — Децентрализованная автономная

организация), а полнота умных контрактов по Тьюрингу в конечном счете привела к радикальному изменению (так называемой "жесткой вилке") в протоколе работы блокчейнов на платформе Ethereum Classic.

Транзакции присваивают биткойны сценарию *блокировки*, определенному в поле ScriptPubKey (см. главу 5). Такой сценарий можно сравнить с запертым на ключ денежным ящиком, а открыть его можно лишь особым ключом. Разумеется, деньги в этом ящике доступны только владельцу такого ключа. Раскрытие денежного ящика осуществляется в поле сценария ScriptSig (см. главу 5). Этим подтверждается право на владение запертым на ключ денежным ящиком, а следовательно, и разрешение на расходование хранящихся в нем денежных средств.

Принцип действия языка Script

Как и большинство языков программирования, язык Script обрабатывает команды по очереди, а команды оперируют элементами стека. В языке Script имеются два возможных типа команд: элементы и операции.

Элементы являются данными. Формально обработка элемента состоит в том, чтобы разместить его в стеке. Элементы представлены байтовыми строками длиной от 1 до 250. Типичным элементом может быть подпись или открытый ключ в формате SEC (рис. 6.1).

Рис. 6.1. Элементы

Операции выполняют определенную обработку данных (рис. 6.2). Они извлекают от нуля и больше элементов из стека обработки и размещают обратно в нем столько же элементов.

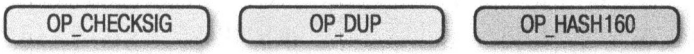

Рис. 6.2. Операции

Типичной является операция `OP_DUP` (рис. 6.3), дублирующая элемент на вершине стека (потребление 0) и размещающая в нем новый элемент (размещение 1).

Рис. 6.3. Операция `OP_DUP` *дублирует элемент на вершине стека*

После того как все команды будут вычислены, элемент на вершине стека должен стать ненулевым, чтобы сценарий был разрешен как достоверный. Если элементы отсутствуют в стеке или на его вершине присутствует нулевой элемент, сценарий будет признан недостоверным. Недостоверный исход означает, что транзакция, включающая в себя сценарий разблокировки, не будет принята в сети.

Примеры операций

Помимо операции `OP_DUP`, имеется немало других операций. Так, операция `OP_HASH160`, схематически представленная на рис. 6.4, выполняет сначала хеш-функцию sha256, а затем — хеш-функцию ripemd160 (называемую также hash160) над элементом на вершине стека (потребление 1) и размещает в нем новый элемент (размещение 1). Глядя на блок-схему данной операции, приведенную на рис. 6.4, следует иметь в виду, что `y = hash160(x)`.

Рис. 6.4. Операция `OP_HASH160` *выполняет сначала хеш-функцию sha256, а затем — хеш-функцию ripemd160 над элементом на вершине стека*

Другой очень важной операцией является `OP_CHECKSIG`, схематически представленная на рис. 6.5. Эта операция потребляет, т.е. извлекает сначала из стека два элемента, первым из которых является открытый ключ, а вторым — подпись, а затем проверяет, пригодна ли подпись для открытого ключа. Если она пригодна, то операция `OP_CHECKSIG` размещает в стеке 1, а иначе — 0.

Рис. 6.5. Операция `OP_CHECKSIG` *проверяет, пригодна ли подпись для открытого ключа*

Программная реализация операций по их кодам

А теперь попробуем реализовать операцию OP_DUP в заданном стеке, как показано ниже. Эта операция просто дублирует элемент на вершине стека.

```
def op_dup(stack):
  if len(stack) < 1:     ❶
    return False
  stack.append(stack[-1])     ❷
  return True
...
OP_CODE_FUNCTIONS = {
  ...
  118: op_dup,     ❸
  ...
}
```

❶ Продублировать необходимо хотя бы один элемент, иначе выполнить операцию по данному коду не удастся.

❷ Именно так дублируется элемент на вершине стека.

❸ 118 = 0x76 — это код операции OP_DUP.

Обратите внимание на то, что по коду данной операции возвращается логическое значение, сообщающее об удачном или неудачном ее завершении. Если операция завершится неудачно, то та же участь автоматически постигнет и сценарий.

Ниже показано, каким образом программируется другая операция, OP_HASH256. Эта операция извлекает элемент из вершины стека, выполняет над ним хеш-функцию hash256 и помещает полученный результат обратно в стек.

```
def op_hash256(stack):
  if len(stack) < 1:
    return False
  element = stack.pop()
  stack.append(hash256(element))
  return True
...
OP_CODE_FUNCTIONS = {
  ...
  170: op_hash256,
  ...
}
```

Упражнение 1

Напишите функцию `op_hash160()`.

Синтаксический анализ полей сценариев

Синтаксический анализ полей ScriptPubKey и ScriptSig осуществляется одинаково. Так, если значение байта находится в пределах от `0x01` до `0x4b` (назовем это значение *n*), то в качестве элемента можно прочитать следующие *n* байтов. В противном случае данный байт представляет операцию, которую придется найти. Ниже перечислены некоторые операции и их байтовые коды.

- `0x00` — `OP_0`
- `0x51` — `OP_1`
- `0x60` — `OP_16`
- `0x76` — `OP_DUP`
- `0x93` — `OP_ADD`
- `0xa9` — `OP_HASH160`
- `0xac` — `OP_CHECKSIG`

Об элементах длиной больше 75 байтов

В связи с изложенным выше у вас может возникнуть вопрос: что произойдет, если длина элемента окажется больше `0x4b` (или `75` в десятичной форме)? Для обработки таких элементов имеются следующие три операции: `OP_PUSHDATA1`, `OP_PUSHDATA2` и `OP_PUSHDATA4`. В частности, операция `OP_PUSHDATA1` означает, что следующий байт содержит количество байтов, которые требуется прочитать для обрабатываемого элемента. Операция `PUSHDATA2` означает, что следующие 2 байта содержат количество байтов, которые требуется прочитать для обрабатываемого элемента. И наконец, операция `OP_PUSHDATA4` означает, что следующие 4 байта содержат количество байтов, которые требуется прочитать для обрабатываемого элемента.

На практике это означает, что если имеется элемент длиной от 76 до 255 байтов включительно, то для его обработки применяется операция `OP_PUSHDATA1` *<длина элемента 1 байт> <элемент>*.

Для обработки элементов длиной от 256 до 520 байтов включительно применяется операция OP_PUSHDATA2 <*длина элемента 2 байта*> <*элемент*>. А элементы длиной больше 520 не допускаются в сети, и потому операция OP_PUSHDATA4 не нужна, хотя операция OP_PUSHDATA4 <*длина элемента 4 байта в прямом порядке их следования, но меньше или ровна 520 байтов*> <*элемент*> все же допустима.

Число, которое меньше 76, можно закодировать с помощью операции OP_PUSHDATA1, число, меньшее 256, — с помощью операции OP_PUSHDATA2, а любое число, меньшее 521, — с помощью операции OP_PUSHDATA4. Но такие транзакции считаются нестандартными, а это означает, что большинство сетевых узлов биткойна (особенно тех, на которых выполняется программное обеспечение Bitcoin Core) не будут их передавать.

Имеется немало других кодов операций, запрограммированных в исходном файле op.py. С полным их перечнем можно ознакомиться по адресу https://en.bitcoin.it/wiki/Script.

Программная реализация синтаксического анализатора и сериализатора сценариев

А теперь, когда известен принцип действия языка Script, можно написать синтаксический анализатор сценариев следующим образом:

```
class Script:

  def __init__(self, cmds=None):
    if cmds is None:
      self.cmds = []
    else:
      self.cmds = cmds     ❶
  ...
  @classmethod
  def parse(cls, s):
    length = read_varint(s)     ❷
    cmds = []
    count = 0
    while count < length:     ❸
      current = s.read(1)     ❹
      count += 1
      current_byte = current[0]     ❺
```

```
    if current_byte >= 1 and current_byte <= 75:     ❻
      n = current_byte
      cmds.append(s.read(n))
      count += n
    elif current_byte == 76:     ❼
      data_length = little_endian_to_int(s.read(1))
      cmds.append(s.read(data_length))
      count += data_length + 1
    elif current_byte == 77:     ❽
      data_length = little_endian_to_int(s.read(2))
      cmds.append(s.read(data_length))
      count += data_length + 2
    else:     ❾
      op_code = current_byte
      cmds.append(op_code)
  if count != length:     ❿
    raise SyntaxError('parsing script failed')
  return cls(cmds)
```

❶ Каждая команда является исполняемым кодом операции или элементом, размещаемым в стеке.

❷ Сериализация сценария всегда начинается с длины всего сценария.

❸ Синтаксический анализ производится до тех пор, пока не будет употреблено нужное количество байтов.

❹ В этом байте определяется код операции или элемент.

❺ Здесь байт преобразуется в целое число.

❻ Для числа в пределах от 1 до 75 включительно известно, что следующие *n* байтов содержат элемент.

❼ Если число равно 76, то выполняется операция `OP_PUSHDATA1`, а в следующем байте указано, сколько байтов должно быть прочитано.

❽ Если число равно 77, то выполняется операция `OP_PUSHDATA2`, а в следующем байте указано, сколько байтов должно быть прочитано.

❾ Получен код операции, который сохраняется.

❿ Должно быть использовано количество байтов, точно равное длине сценария, а иначе происходит ошибка.

Аналогичным образом можно написать и сериализатор сценариев:

```
class Script:
...
  def raw_serialize(self):
```

```
    result = b''
    for cmd in self.cmds:
      if type(cmd) == int:    ❶
        result += int_to_little_endian(cmd, 1)
      else:
        length = len(cmd)
      if length < 75:    ❷
        result += int_to_little_endian(length, 1)
      elif length > 75 and length < 0x100:    ❸
        result += int_to_little_endian(76, 1)
        result += int_to_little_endian(length, 1)
      elif length >= 0x100 and length <= 520:    ❹
        result += int_to_little_endian(77, 1)
        result += int_to_little_endian(length, 2)
      else:    ❺
        raise ValueError('too long an cmd')
        result += cmd
    return result

  def serialize(self):
    result = self.raw_serialize()
    total = len(result)
    return encode_varint(total) + result    ❻
```

❶ Если команда обозначена целым числом, значит, это код операции.

❷ Если длина элемента находится в пределах от 1 до 75 включительно, она кодируется одним байтом.

❸ Для любого элемента длиной от 76 до 255 сначала задается код операции `OP_PUSHDATA1`, а затем длина элемента кодируется одним байтом, после которого следует сам элемент.

❹ Для любого элемента длиной от 256 до 520 сначала задается код операции `OP_PUSHDATA2`, затем длина элемента кодируется двумя байтами в прямом порядке их следования, а после них следует сам элемент.

❺ Любой элемент длиной больше 520 байтов не подлежит сериализации.

❻ Сериализация сценария начинается с длины всего сценария.

Следует заметить, что приведенный выше синтаксический анализатор и сериализатор уже применялись в главе 5 для синтаксического анализа и сериализации полей ScriptSig и ScriptPubKey.

Объединение полей сценариев

Объект типа `Script` представляет набор команд, которые требуется вычислить. А для того, чтобы вычислить сценарий, необходимо объединить поля ScriptPubKey и ScriptSig. Запертый на ключ денежный ящик (сценарий ScriptPubKey) и отпирающий его механизм (сценарий ScriptSig) являются *разными* транзакциями. В частности, денежный ящик служит местом для прихода биткойнов, а отпирающий его механизм или сценарий — местом для расхода биткойнов. Ввод в расходной транзакции *указывает на приходную транзакцию*. По существу, возникает такая же ситуация, как и показанная на рис. 6.6.

Рис. 6.6. Объединение сценариев ScriptPubKey и ScriptSig

Сценарий ScriptSig разблокирует сценарий ScriptPubKey, и поэтому требуется какой-то механизм для объединения обоих сценариев. Чтобы вычислить оба сценария вместе, необходимо извлечь команды из полей ScriptSig и ScriptPubKey и объединить их, как показано на рис. 6.6. При этом команды из поля ScriptSig должны предшествовать всем командам из поля ScriptPubKey. Команды обрабатываются по очереди до тех пор, пока необработанных команд больше не останется или же объединенный сценарий завершится неудачно.

Программная реализация объединенного набора команд

Чтобы вычислить сценарий, необходимо объединить содержимое полей ScriptSig и ScriptPubKey в единый набор команд и затем выполнить эти команды. Ниже показано, каким образом сценарии объединяются непосредственно в коде.

```
class Script:
...
  def __add__(self, other):
    return Script(self.cmds + other.cmds)     ❶
```

❶ Здесь команды из обоих исходных сценариев объединяются в единый набор, для которого создается новый объект типа `Script`.

Такой возможностью объединять сценарии для их вычисления мы воспользуемся далее в этой главе.

Стандартные сценарии

В биткойне имеются самые разные типы стандартных сценариев, включая следующие.

p2pk

Оплата по открытому ключу.

p2pkh

Оплата по хешу открытого ключа.

p2sh

Оплата по хешу сценария.

p2wpkh

Оплата по хешу открытого ключа с раздельным заверением.

p2wsh

Оплата по хешу сценария с раздельным заверением.

Адресами служат известные шаблоны сценариев, подобные перечисленным выше, а в криптовалютных кошельках известно, как интерпретировать различные типы адресов (p2pkh, p2sh, p2wpkh) и создавать подходящие сценарии типа ScriptPubKey. Все рассматриваемые здесь примеры сценариев имеют конкретный формат адреса (Base58, Bech32), чтобы по нему можно было производить оплату из криптовалютных кошельков.

p2pk

Сценарий оплаты по открытому ключу (p2pk) широко применялся с самого зарождения биткойна. Большинство биткойнов, которые, как считалось, принадлежали Сатоши Накомото, присутствуют в UTXO для p2pk, т.е. на

выводах транзакций, где сценарии ScriptPubKey имеют форму p2pk. У сценария p2pk имеется ряд ограничений, подробнее объясняемых далее, в разделе "Затруднения, связанные с p2pk", но сначала рассмотрим принцип действия этого сценария.

Из главы 3 вам должно быть известно о подписании и верификации по алгоритму ECDSA. Для верификации подписи по алгоритму ECDSA требуются сообщение (z), открытый ключ (P) и сама подпись (r и s). В сценарии p2pk биткойны отсылаются по открытому ключу, а владелец секретного ключа может разблокировать или отправить биткойны, создав подпись. Сценарий ScriptPubKey транзакции устанавливает контроль владельца секретного ключа над присвоенными биткойнами.

Указать, куда направляются биткойны, — задача сценария ScriptPubKey, который служит в качестве денежного ящика для приема биткойнов. Сценарий ScriptPubKey для p2pk выглядит так, как показано на рис. 6.7.

```
410411db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5cb2e0eaddfb84c
cf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3ac

        - 41 - Длина открытого ключа
        - 0411...a3 - <открытый ключ>
        - ac - OP_CHECKSIG
```

Рис. 6.7. Сценарий ScriptPubKey для оплаты по открытому ключу (p2pk)

Обратите внимание на операцию `OP_CHECKSIG`, поскольку это очень важно. Сценарий ScriptSig относится к той части рассматриваемого здесь сценария, которая разблокирует полученные биткойны. Открытый ключ может быть в сжатом или несжатом формате, хотя на начальной стадии развития биткойна, когда сценарий p2pk играл более заметную роль, применялся только несжатый формат (см. главу 4).

Для p2pk сценарий ScriptSig, требующийся для разблокировки соответствующего открытого ключа из сценария ScriptPubKey, состоит из подписи и одного байта с хешем данной подписи, как показано на рис. 6.8.

```
47304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548ab5fb8cd410220181
522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d0901

        - 47 - Длина подписи
        - 3044...01 - <подпись>
```

Рис. 6.8. Сценарий ScriptSig для оплаты по открытому ключу (p2pk)

Совместно сценарии ScriptPubKey и ScriptSig образуют набор команд, который выглядит так, как показано на рис. 6.9.

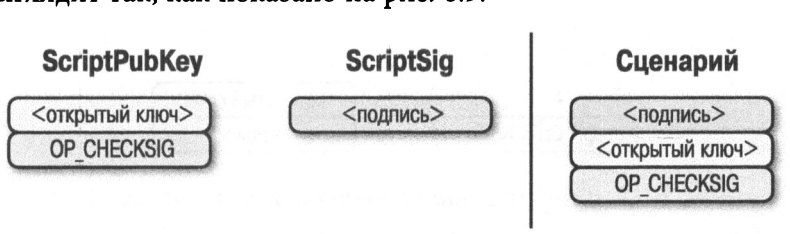

Рис. 6.9. Объединенный сценарий p2pk

В двух столбцах на рис. 6.10 схематически показаны команды Script и стек элементов. По окончании обработки элемент на вершине стека должен стать ненулевым, чтобы считать сценарий ScriptSig достоверным. Команды Script обрабатываются по очереди. Как показано на рис. 6.10, сначала обрабатываются команды, объединенные на рис. 6.9.

Рис. 6.10. Начальное состояние сценария p2pk

Первой командой оказывается подпись, которая является элементом. Это данные, которые размещаются в стеке (рис. 6.11).

Рис. 6.11. Первая стадия выполнения сценария p2pk

Второй командой оказывается открытый ключ, который также является элементом. И это снова данные, которые размещаются в стеке (рис. 6.12).

Рис. 6.12. Вторая стадия выполнения сценария p2pk

В операции `OP_CHECKSIG` сначала извлекаются из стека две команды (открытый ключ и подпись), а затем проверяется их достоверность для данной транзакции. В итоге операция `OP_CHECKSIG` разместит в стеке 1, если подпись окажется достоверной, а иначе — 0. Так, если подпись достоверна для данного открытого ключа, то возникает ситуация, наглядно показанная на рис. 6.13.

Рис. 6.13. Третья стадия выполнения сценария p2pk

Таким образом, обработка всех команд завершена, а в стеке остался один элемент. Данный сценарий теперь считается достоверным, поскольку на вершине стека находится ненулевой элемент (ведь 1 — это, определенно, не 0).

Если бы подпись в данной транзакции оказалась недостоверной, то в результате выполнения операции `OP_CHECKSIG` на вершине стека оказался бы нулевой элемент, и на этом обработка сценария завершилась бы, как показано на рис. 6.14. Если же на вершине стека находится нулевой элемент, то недостоверным считается объединенный сценарий, а следовательно, и транзакция с таким сценарием ScriptSig на вводе.

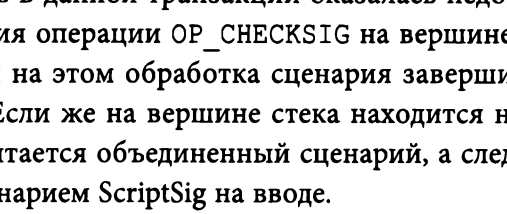

Рис. 6.14. Конечное состояние сценария p2pk

В объединенном сценарии проверяется достоверность подписи, и если она окажется недостоверной, то сценарий завершится неудачно. Сценарий ScriptSig разблокирует открытый ключ из сценария ScriptPubKey лишь в том случае, если подпись окажется достоверной для этого ключа. Иными словами, сделать подпись по сценарию ScriptSig может лишь тот, кому известен секретный ключ.

Выясним попутно, откуда сценарий ScriptPubKey получил свое название. Дело в том, что открытый ключ в несжатом формате SEC является основной командой в сценарии ScriptPubKey для p2pk (другой его командой является операция `OP_CHECKSIG`). Аналогично сценарий ScriptSig называется так потому, что для p2pk он содержит подпись в формате DER.

Программная реализация вычисления сценариев

Теперь мы можем запрограммировать вычисление сценариев. Для этого нам придется перебрать каждую команду и проверить сценарий на достоверность. Итак, требуется, чтобы выполнялись следующие действия:

```
>>> from script import Script
>>> z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e\
29307032997a838a3d
>>> sec = bytes.fromhex('04887387e452b8eacc4acfde10d9aaf\
7f6d9a0f975aabb10d006e\4da568744d06c61de6d95231cd89026e286\
df3b6ae4a894a3378e393e93a0f45b666329a0ae34')
>>> sig = bytes.fromhex('3045022000eff69ef2b1bd93a66ed5219\
add4fb51e11a840f404876325a1e8ffe0529a2c022100c7207fee197d27\
c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab601')
>>> script_pubkey = Script([sec, 0xac])     ❶
>>> script_sig = Script([sig])
>>> combined_script = script_sig + script_pubkey     ❷
>>> print(combined_script.evaluate(z))     ❸
True
```

❶ Сценарий ScriptPubkey для p2pk состоит из открытого ключа в формате SEC и операции `OP_CHECKSIG` с кодом `0xac` или 172.

❷ Эта операция допустима потому, что ранее был определен метод `__add__()`.

❸ Здесь требуется вычислить команды и проверить сценарий на достоверность.

Ниже приведен метод, который можно применить к *объединенному* набору команд (сочетанию сценария ScriptPubKey из предыдущей транзакции и сценария ScriptSig из текущей транзакции).

```
from op import OP_CODE_FUNCTIONS, OP_CODE_NAMES
...
class Script:
...
  def evaluate(self, z):
    cmds = self.cmds[:]     ❶
    stack = []
    altstack = []
    while len(cmds) > 0:     ❷
      cmd = cmds.pop(0)
      if type(cmd) == int:
        operation = OP_CODE_FUNCTIONS[cmd]     ❸
        if cmd in (99, 100):     ❹
          if not operation(stack, cmds):
            LOGGER.info('bad op: {}'.format(OP_CODE_NAMES[cmd]))
            return False
          elif cmd in (107, 108):     ❺
            if not operation(stack, altstack):
              LOGGER.info('bad op: {}'.format(OP_CODE_NAMES[cmd]))
              return False
            elif cmd in (172, 173, 174, 175):     ❻
               if not operation(stack, z):
                 LOGGER.info('bad op: {}'.format(OP_CODE_NAMES[cmd]))
                 return False
            else:
              if not operation(stack):
                LOGGER.info('bad op: {}'.format(OP_CODE_NAMES[cmd]))
                return False
      else:
        stack.append(cmd)     ❼
    if len(stack) == 0:
      return False     ❽
    if stack.pop() == b'':
      return False     ❾
    return True     ❿
```

❶ Сделать копию списка команд, поскольку он изменится.

❷ Выполнять команды вплоть до исчерпания их списка.

❸ Функция, выполняющая код операции, находится в массиве OP_CODE_FUNCTIONS (например, OP_DUP, OP_CHECKSIG и т.д.).

❹ Числа 99 и 100 соответствуют кодам операций OP_IF и OP_NOTIF. Они требуют манипулирования массивом cmds, исходя из того, какой именно элемент находится на вершине стека.

❺ Числа 107 и 108 соответствуют кодам операций OP_TOALTSTACK и OP_FROMALTSTACK. Эти операции размещают и извлекают элементы из "альтернативного" стека, называемого altstack.

❻ Числа 172, 173, 174 и 175 соответствуют операциям OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG и OP_CHECKMULTISIGVERIFY, которым требуется хеш подписи (*z*; см. главу 3) для ее проверки на достоверность.

❼ Если команда не является кодом операции, то она является элементом, а следовательно, этот элемент размещается в стеке.

❽ Если стек оказывается пустым по окончании обработки всех команд, сценарий завершается неудачно, и поэтому возвращается логическое значение False.

❾ Если на вершине стека находится пустая байтовая строка (а именно в таком виде в стеке хранится 0), то и в этом случае сценарий завершается неудачно, а следовательно, возвращается логическое значение False.

❿ Любой другой результат означает, что сценарий оказался достоверным.

Безопасное вычисление сценариев

Приведенный выше код несколько обманчив, поскольку объединенный сценарий именно так не выполняется. В частности, сценарий ScriptSig вычисляется отдельно от сценария ScriptPubKey для того, чтобы операции из сценария ScriptSig не оказывали никакого влияния на команды из сценария ScriptPubKey. Кроме того, стек сохраняется после вычисления всех команд из сценария ScriptSig, а затем команды из сценария ScriptPubKey вычисляются отдельно со стеком из первой стадии выполнения.

Внутреннее представление элементов в стеке

Обозначение элементов в стеке числами вроде 0 или 1, а иногда и байтовыми строками вроде подписи в формате DER или открытого ключа в формате SEC может вызвать недоумение. Внутренне все они представлены байтами,

хотя некоторые из них интерпретируются как числа, обозначающие определенные коды операций. Например, число 1 хранится в стеке в виде байта 01, 2 — в виде байта 02, 999 — в виде байта e703 и т.д. Любая байтовая строка интерпретируется как число, представленное последовательностью байтов в прямом порядке их следования, и предназначенное для обозначения кода арифметической операции. Целое число 0 хранится в стеке *не* в виде байта 00, а как пустая байтовая строка.

В приведенном ниже фрагменте кода из исходного файла `op.py` поясняется суть дела. В частности, числа, размещаемые в стеке, кодируются в байты, а числа, извлекаемые из стека, декодируются из байтов, когда требуется снова получить числовое значение.

```
def encode_num(num):
  if num == 0:
    return b''
  abs_num = abs(num)
  negative = num < 0
  result = bytearray()
  while abs_num:
    result.append(abs_num & 0xff)
    abs_num >>= 8
  if result[-1] & 0x80:
    if negative:
      result.append(0x80)
    else:
      result.append(0)
    elif negative:
      result[-1] |= 0x80
    return bytes(result)

def decode_num(element):
  if element == b'':
    return 0
  big_endian = element[::-1]
  if big_endian[0] & 0x80:
    negative = True
    result = big_endian[0] & 0x7f
  else:
    negative = False
    result = big_endian[0]
  for c in big_endian[1:]:
    result <<= 8
    result += c
  if negative:
    return -result
```

```
  else:
    return result

def op_0(stack):
  stack.append(encode_num(0))
  return True
```

Упражнение 2

Напишите свою версию функции `op_checksig()` из исходного файла `op.py`.

Затруднения, связанные с p2pk

Оплата по открытому ключу вполне очевидна в том смысле, что, имея в своем распоряжении открытый ключ, всякий может отправить биткойны по этому ключу вместе с подписью, которую может сделать только владелец секретного ключа. И хотя такой сценарий вполне работоспособен, он все же вызывает некоторые затруднения.

Во-первых, открытые ключи длинны. Как упоминалось в главе 4, открытые точки на эллиптической кривой secp256k1 занимают 33 байта в сжатом формате SEC и 65 байтов в несжатом формате SEC. Но, к сожалению, эти байты неудобочитаемы в исходном виде, а большинство кодировок не воспроизводят байты в определенных пределах, где находятся управляющие символы, знаки новой строки, табуляции и им подобные. Вместо этого данные в формате SEC, как правило, кодируются в шестнадцатеричной форме, где их длина удваивается (по 4 бита на каждый символ вместо 8). В итоге открытые точки оказываются длиной 66 и 130 символов в сжатом и несжатом форматах SEC соответственно, что существенно больше, чем длина большинства идентификаторов (например, длина имени пользователя на веб-сайте обычно не больше 20 символов). Кроме того, в первоначальных транзакциях биткойна не применялись сжатые версии, и поэтому длина каждого адреса в шестнадцатеричном виде составляла 130 символов! Записывать такие адреса людям было неудобно и нелегко, а тем более передавать их голосом.

Тем не менее сценарий p2pk первоначально применялся для оплаты по межсетевым адресам и выработки добываемой криптовалюты. Что касается оплаты по межсетевым адресам, то IP-адреса запрашивались по их открытым ключам, которые передавались между компьютерами, а это означало, что общение между людьми совсем не обязательно вызывало трудности. А вот для

выработки добываемой криптовалюты общение между людьми все же требовалось. Впрочем, система оплаты по межсетевым адресам была упразднена, поскольку была небезопасна и подвержена атакам со стороны посредника.

Зачем Сатоши Накомото пользовался несжатым форматом SEC

На первый взгляд, пользоваться несжатым форматом SEC не имеет особого смысла для биткойна, в котором свободное место в блоке на вес золота. Зачем же Сатоши Накомото им пользовался? Дело в том, что он пользовался библиотекой OpenSSL для преобразований в формат SEC, а в то время, когда он разрабатывал технологию биткойна (около 2008 года), сжатый формат SEC был не очень хорошо документирован в библиотеке OpenSSL. Поэтому можно лишь догадываться, почему Сатоши Накомото выбрал именно несжатый формат SEC. Когда же Питер Виль (знаток криптографии) обнаружил, что в библиотеке OpenSSL поддерживается и сжатый формат SEC, вслед за ним многие стали пользоваться этим форматом в биткойне.

Во-вторых, длина открытых ключей вызывает менее очевидное затруднение: множество UTXO укрупняется, поскольку открытые ключи приходится хранить и индексировать, чтобы проверять, израсходованы ли выводы транзакции. А для этого требуются дополнительные ресурсы со стороны полных узлов сети биткойна.

И в-третьих, открытые ключи хранятся в поле ScriptPubKey, а следовательно, они общеизвестны. Это означает, что когда-нибудь алгоритм шифрования ECDSA будет взломан, и соответствующие выводы транзакций могут быть украдены. Например, квантовое вычисление позволяет существенно сократить время вычисления для алгоритмов RSA и ECDSA, и поэтому для повышения безопасности требуется нечто большее, чем меры защиты выводов транзакций. Впрочем, это не такая уж и большая угроза, поскольку алгоритм ECDSA находит применение во многих других областях, помимо биткойна, а его взлом оказал бы отрицательное влияние во всех подобных областях.

Разрешение затруднений средствами p2pkh

Оплата по хешу открытого ключа (p2pkh) является альтернативным сценарием, обладающим следующими преимуществами над p2pk.

1. Более короткие адреса.
2. Дополнительная защита с помощью алгоритмов хеширования sha256 и ripemd160.

Адреса оказываются более короткими потому, что для их формирования применяются алгоритмы хеширования sha256 и ripemd160. Для этого соответствующие хеш-функции, sha256 и ripemd160, выполняются последовательно, после чего вызывается хеш-функция hash160. В результате выполнения хеш-функции hash160 получаются 160 бит или 20 байтов, в которые и кодируется адрес.

В итоге получается следующий закодированный адрес, который можно наблюдать в сети биткойна и который упоминался в главе 4:

```
1PMycacnJaSqwwJqjawXBErnLsZ7RkXUAs
```

Этот адрес кодируется 20 байтами, которые выглядят в шестнадцатеричной форме следующим образом:

```
f54a5851e9372b87810a8e60cdd2e7cfd80b6e31
```

Эти 20 байтов получаются в результате выполнения хеш-функции hash160 над открытым ключом, представленным ниже в сжатом формате SEC.

```
0250863ad64a87ae8a2fe83c1af1a8403cb53f53e486d8511dad8a04887e5b2352
```

В связи с тем, что сценарий p2pkh обеспечивает большую краткость и безопасность, его применение значительно сократилось после 2010 года, хотя он до сих пор находит полную поддержку.

p2pkh

Оплата по хешу открытого ключа применялась еще на ранней стадии развития биткойна, хотя и не в такой степени, как p2pk. Сценарий блокировки ScriptPubKey для p2pkh выглядит так, как показано на рис. 6.15.

```
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac

- 76 - OP_DUP
- a9 - OP_HASH160
- 14 - Длина <хеша>
- bc3d...da - <хеш>
- 88 - OP_EQUALVERIFY
- ac - OP_CHECKSIG
```

Рис. 6.15. Сценарий ScriptPubKey для оплаты по хешу открытого ключа (p2pkh)

Как и в p2pk, здесь выполняется операция `OP_CHECKSIG`, а операция `OP_HASH160` придает вешний вид. Но в отличие от p2pk открытый ключ

представлен здесь не в формате SEC, а в хешированном 20-байтовом виде. Кроме того, здесь введена новая операция OP_EQUALVERIFY.

Сценарий блокировки ScriptSig для p2pkh выглядит так, как показано на рис. 6.16.

```
483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf2
1320b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f678
01c31967743a9c8e10615bed01210349fc4e631e3624a545de3f89f5d868
4c7b8138bd94bdd531d2e213bf016b278a

        - 48 - Длина <подписи>
        - 30...01 - <подпись>
        - 21 - Длина <открытого ключа>
        - 0349...8a - <открытый ключ>
```

Рис. 6.16. Сценарий ScriptSig для оплаты по хешу открытого ключа (p2pkh)

Как и в p2pk, сценарий ScriptSig содержит подпись в формате DER. Но в отличие от p2pk сценарий ScriptSig содержит также открытый ключ в формате SEC. Главное различие сценариев ScriptSig для p2pk и для p2pkh заключается в том, что открытый ключ формата SEC перенесен из поля ScriptPubKey в поле ScriptSig.

Совместно сценарии ScriptPubKey и ScriptSig образуют набор команд, как показано на рис. 6.17.

Рис. 6.17. Объединенный сценарий p2pkh

На данном этапе команды из объединенного сценария обрабатываются по очереди. А начинается он с команд, приведенных на рис. 6.18.

Первые две команды в этом сценарии являются элементами, а потому размещаются в стеке (рис. 6.19).

В операции OP_DUP выполняется дублирование элемента, находящегося на вершине стека. В итоге открытый ключ дублируется (рис. 6.20).

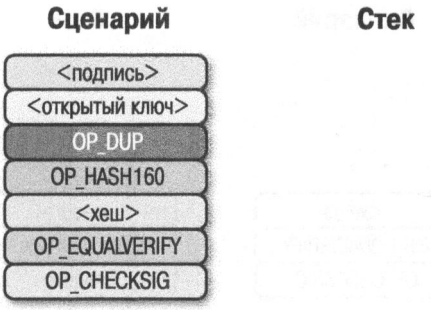

Рис. 6.18. Начальное состояние сценария p2pkh

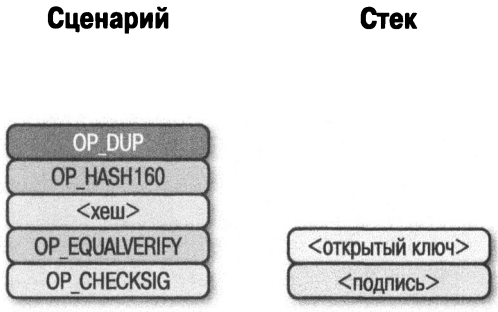

Рис. 6.19. Первая стадия выполнения сценария p2pkh

Рис. 6.20. Вторая стадия выполнения сценария p2pkh

В операции `OP_HASH160` над элементом, который берется из вершины стека, сначала выполняется хеш-функция hash160, а затем — хеш-функции sha256 и ripemd160. В итоге получается 20-байтовый хеш (рис. 6.21).

20-байтовый хеш является элементом и поэтому размещается в стеке (рис. 6.22).

Рис. 6.21. Третья стадия выполнения сценария p2pkh

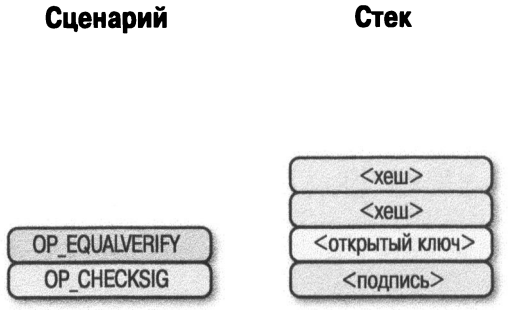

Рис. 6.22. Четвертая стадия выполнения сценария p2pkh

Теперь очередь доходит до операции `OP_EQUALVERIFY`. В этой операции используются два верхних элемента в стеке и проверяется их равенство. Если они равны, то выполнение данного сценария продолжается. А если они не равны, выполнение данного сценария сразу же прекращается, завершаясь неудачно. Здесь предполагается, что сравниваемые элементы равны, что приводит к состоянию данного сценария, показанному на рис. 6.23.

Рис. 6.23. Пятая стадия выполнения сценария p2pkh

Итак, мы дошли до такой же стадии, как и при выполнении операции OP_CHECKSIG при обработке сценария p2pk. И здесь предполагается, что подпись достоверна (рис. 6.24).

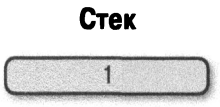

Рис. 6.24. Конечное состояние сценария p2pkh

Имеются два условия неудачного завершения сценария p2pkh. Если в сценарии ScriptSig предоставляется открытый ключ, *не* хешированный хеш-функцией hash160 в 20-байтовый хеш в сценарии ScriptPubKey, то сценарий p2pkh завершится неудачно при выполнении операции OP_EQUALVERIFY (рис. 6.22). Другое условие неудачного завершения сценария p2pkh состоит в том, что в сценарии ScriptSig имеется недостоверная подпись, несмотря на наличие открытого ключа, хешированного хеш-функцией hash160 в 20-байтовый хеш в сценарии ScriptPubKey. В итоге вычисление объединенного сценария завершится нулем, а следовательно, неудачно.

Именно поэтому такого рода сценарий называется оплатой по *хешу* открытого ключа. Ведь в поле ScriptPubKey содержится 20-байтовый хеш, полученный из открытого ключа с помощью хеш-функции *hash160*, а не сам открытый ключ. Биткойны блокируются по *хешу* открытого ключа, а тот, кто их тратит, отвечает за раскрытие открытого ключа как части построения подписи по сценарию ScriptSig.

Главное преимущество p2pkh заключается в том, что открытый ключ из сценария ScriptPubKey оказывается более коротким (длиной всего лишь 25 байтов). А его вору придется не только решить задачу дискретного логарифмирования в алгоритме ECDSA, но и найти способ обнаружения прообразов обоих хешей, сформированных хеш-функциями ripemd160 и sha256.

Построение произвольных сценариев

Следует, однако, иметь в виду, что в качестве сценария может служить любая программа. И как упоминалось ранее, Script — это язык составления умных контрактов, позволяющий блокировать биткойны самыми разными способами. Так, на рис. 6.25 приведен пример сценария ScriptPubKey.

А на рис. 6.26 приведен пример сценария ScriptSig для разблокировки открытого ключа из сценария ScriptPubKey.

```
55935987

55 - OP_5
93 - OP_ADD
59 - OP_9
87 - OP_EQUAL
```

Рис. 6.25. Пример сценария ScriptPubKey

```
54

54 - OP_4
```

Рис. 6.26. Пример сценария ScriptSig

Объединенный в итоге сценарий выглядит так, как показано на рис. 6.27.

Рис. 6.27. Пример объединенного сценария

Вычисление объединенного сценария начнется так, как показано на рис. 6.28.

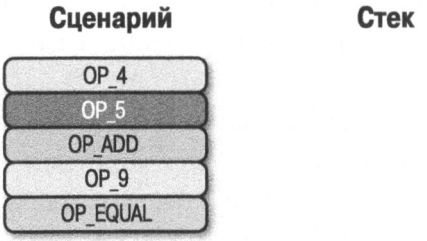

Рис. 6.28. Начальное состояние объединенного сценария

При выполнении операции `OP_4` в стеке будет размещено число 4 (рис. 6.29).

Аналогично при выполнении операции `OP_5` в стеке будет размещено число 5 (рис. 6.30).

При выполнении операции `OP_ADD` из стека будут извлечены и сложены вместе два элемента, а полученная в итоге сумма помещена обратно в стек (рис. 6.31).

Рис. 6.29. Первая стадия выполнения объединенного сценария

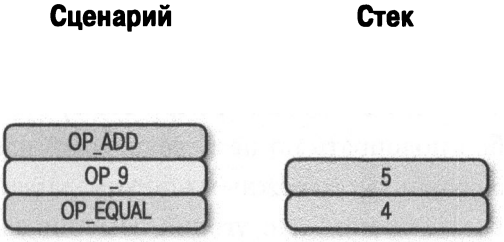

Рис. 6.30. Вторая стадия выполнения объединенного сценария

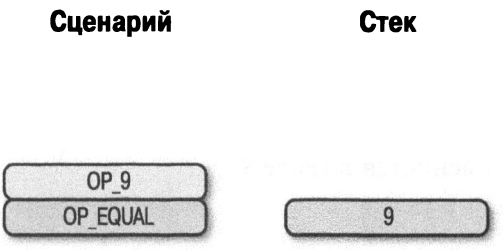

Рис. 6.31. Третья стадия выполнения объединенного сценария

При выполнении операции `OP_9` в стеке будет размещено число 9 (рис. 6.32).

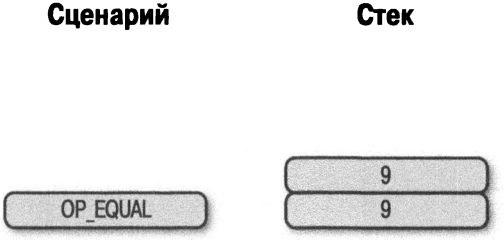

Рис. 6.32. Четвертая стадия выполнения объединенного сценария

При выполнении операции OP_EQUAL из стека будут извлечены два элемента, а обратно в него будет помещена 1, если оба элемента равны, или 0, если они не равны (рис. 6.33).

Рис. 6.33. Пятая стадия выполнения объединенного сценария

Следует, однако, иметь в виду, что в данном примере выявить содержимое сценария ScriptSig не составит особого труда, поскольку в нем фактически отсутствует подпись. Но в итоге сценарий ScriptPubKey окажется уязвимым для всякого, кто способен подобрать по нему секретный ключ. Такой сценарий ScriptPubKey можно сравнить с денежным ящиком, запертым на ненадежный замок, который может взломать кто угодно. Именно по этой причине для большинства транзакций требуется наличие подписи в сценарии ScriptSig.

Как только вывод неизрасходованных транзакций UTXO будет израсходован, включен в блок и защищен подтверждением работы, биткойны будут заблокированы по другому сценарию ScriptPubKey, и потратить их в дальнейшем будет нелегко. А тому, кто попытается израсходовать уже израсходованные биткойны, придется предоставить подтверждение работы, что обойдется ему недешево, как поясняется в главе 9.

Упражнение 3

Создайте сценарий ScriptSig, способный разблокировать открытый ключ по следующему сценарию ScriptPubKey:

```
767695935687
```

Имейте в виду, что в операции OP_MUL умножаются два верхних элемента в стеке, а коды в сценарии ScriptPubKey обозначают следующие операции.

- 56 = OP_6
- 76 = OP_DUP
- 87 = OP_EQUAL
- 93 = OP_ADD
- 95 = OP_MUL

Польза сценариев

Задание в предыдущем упражнении было несколько обманчивым, поскольку операция OP_MUL больше не разрешается в биткойне. В версии 0.3.5 биткойна запрещено немало операций — даже те, которые были способны сделать сеть биткойна уязвимой даже в минимальной степени.

И это только к лучшему, поскольку большинство функциональных возможностей Script не особенно используются. Но для сопровождения программного обеспечения это не совсем хорошо, поскольку малоиспользуемый код все равно приходится сопровождать. Упрощение и отказ от использования некоторых функциональных возможностей можно рассматривать как способ повысить безопасность биткойна. Этим данный проект заметно отличается от других аналогичных проектов, в которых предпринимаются попытки расширить функциональные возможности языков составления умных контрактов, что нередко приводит к большему разнообразию видов атак.

Упражнение 4

Выясните назначение следующего сценария:

```
6e879169a77ca787
```

Коды в этом сценарии обозначают следующие операции.

- 69 = OP_VERIFY
- 6e = OP_2DUP
- 7c = OP_SWAP
- 87 = OP_EQUAL
- 91 = OP_NOT
- a7 = OP_SHA1

Воспользуйтесь методом `Script.parse()` и выясните назначение отдельных операций по их кодам, обратившись по адресу `https://en.bitcoin.it/wiki/Script`.

Пиньята для алгоритма SHA-1

В 2013 году Питер Тодд создал сценарий, очень похожий на сценарий из упражнения 4, вложив в него немного биткойнов, чтобы материально

заинтересовать других находить коллизии хеш-функций. В итоге пожертвования достигли 2,49153717 биткойна, а когда компания Google в феврале 2017 года фактически обнаружила первую коллизию хеш-функций в алгоритме SHA-1 (`https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`), этот сценарий быстро окупился. Сумма на выводе транзакции составила 2,48 биткойна, что на тот момент было равнозначно 2848,88 доллара США. Питер создал и другие сценарии типа пиньяты[1] для хеш-функций sha256, hash256 и hash160, чтобы повысить материальную заинтересованность в поиске их коллизий.

Заключение

В этой главе были рассмотрены внутренний механизм и принцип действия сценариев на языке Script, используемом для составления умных контрактов. А теперь можно перейти к обсуждению вопросов создания и проверки достоверности транзакций.

[1] Мексиканская полая игрушка из папье-маше, в данном случае — нечто вроде копилки. — *Примеч. ред.*

ГЛАВА 7

Создание и проверка достоверности транзакций

Одной из самых трудных задач программирования биткойна является проверка достоверности транзакций, а другой — их создание. В этой главе поясняются конкретные шаги для того и другого. Проработав материал этой главы, вы сможете самостоятельно создать и разослать транзакцию по всей сети testnet.

Проверка достоверности транзакций

Получая транзакции, каждый сетевой узел проверяет ее на соответствие правилам, установленным в сети биткойна. Этот процесс называется *проверкой достоверности транзакций*. В ходе этого процесса сетевой узел проверяет следующие условия.

1. Израсходованы ли ранее вводы транзакции.
2. Превышает ли сумма вводов сумму выводов и не равны ли они.
3. Удачно ли разблокирован в сценарии ScriptSig открытый ключ из сценария ScriptPubKey.

Проверка первого из перечисленных выше условий позволяет исключить двойное расходование. В частности, любой израсходованный ввод, т.е. такой ввод, который включен в блокчейн, не может быть израсходован снова.

Проверка второго условия позволяет убедиться, что новые биткойны не созданы. Это не касается специальной, так называемой монетизирующей, транзакции, более подробно рассматриваемой в главе 9.

И наконец, проверка третьего условия позволяет убедиться в достоверности объединенного сценария. В подавляющем большинстве транзакций это

означает, что одна или больше подписей в сценарии ScriptSig действительны. А теперь рассмотрим порядок проверки этих условий.

Проверка расходования вводов транзакции

Чтобы исключить двойное расходование, в сетевом узле проверяется, существует ли каждый ввод и не израсходован ли он. Такая проверка может быть произведена в любом полном узле в ходе анализа множества UTXO (см. главу 5). Из самой транзакции нельзя выяснить, расходится ли она дважды, подобно тому, как нельзя проверить перерасход средств, глядя на именной чек. Единственный способ выявить двойное расходование — получить доступ к множеству UTXO, для чего придется произвести расчет, исходя из всего множества транзакций.

В биткойне можно выяснить, израсходован ли ввод дважды, отследив выводы неизрасходованных транзакций UTXO. Так, если вывод транзакции принадлежит множеству UTXO, он существует и *не* расходуются дважды. Если же транзакция проходит остальные проверки на достоверность, то все ее вводы удаляются из множества UTXO. "Тонкие" клиенты, не имеющие доступа к блокчейну, вынуждены доверять другим узлам получение многих сведений, в том числе и о том, был ли ввод израсходован. В полном узле совсем не трудно проверить расходование ввода, но для этого "тонкому" клиенту придется добыть сведения у кого-нибудь другого.

Проверка суммы вводов относительно суммы выводов транзакции

В узлах проверяется также, превышает ли сумма вводов сумму выводов и не равны ли они. Тем самым гарантируется, что транзакция не создает новые биткойны. Единственным исключением из этого правила является монетизирующая транзакция, более подробно рассматриваемая в главе 9. Но поскольку на вводах отсутствует поле суммы, его придется искать в блокчейне. И в этом случае полным узлам доступны суммы, связанные с неизрасходованным вводом, тогда как "тонким" клиентам придется обращаться к полным узлам, чтобы добыть такие сведения.

О том, как рассчитывается плата за транзакцию, речь уже шла в главе 5. Проверить, превышает ли сумма вводов сумму выводов и не равны ли они, — это все равно, что проверить, не отрицательна ли сумма оплаты транзакции, т.е. не создаются ли новые биткойны. Возвращаясь к последнему упражнению

из главы 5, напомним, каким образом определяется метод `fee()` для расчета платы за транзакцию.

```
class Tx:
...
  def fee(self):
    '''Возвращает плату за данную транзакцию в сатоши'''
    input_sum, output_sum = 0, 0
    for tx_in in self.tx_ins:
      input_sum += tx_in.value(self.testnet)
    for tx_out in self.tx_outs:
      output_sum += tx_out.amount
    return input_sum - output_sum
```

Проверить, предпринимается ли в данной транзакции попытка создать новые биткойны, можно следующим образом:

```
>>> from tx import Tx
>>> from io import BytesIO
>>> raw_tx = ('0100000001813f79011acb80925dfe69b3def355fe914bd\
1d96a3f5f71bf8303c6a989c7d1000000006b483045022100ed81ff192e75a3\
fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955\
c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349\
fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278\
afeffffff02a135ef01000000001976a914bc3b654dca7e56b04dca18f2566\
cdaf02e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e3321667\
02cb75f40df79fea1288ac19430600')
>>> stream = BytesIO(bytes.fromhex(raw_tx))
>>> transaction = Tx.parse(stream)
>>> print(transaction.fee() >= 0)     ❶
True
```

❶ Такой способ вполне пригоден в Python (см. ниже раздел "Ошибка превышения предельного значения").

Если сумма оплаты транзакции отрицательна, значит, сумма выводов (`output_sum`) больше суммы вводов (`input_sum`). Иными словами, в данной транзакции так или иначе предпринимается попытка создать биткойны.

Ошибка превышения предельного значения

В 2010 году была произведена транзакция, в которой было создано 184 миллиарда биткойнов. Это произошло потому, что в языке C++ поле суммы относится к целочисленному типу *со знаком*, а не *без знака*. Это означает, что значение в этом поле могло быть отрицательным!

Столь хитро составленная транзакция прошла все проверки, в том числе и ту, в ходе которой проверяется, создаются ли новые биткойны, но это произошло только потому, что суммы выводов превысили максимальное числовое значение: $2^{64} \sim 1.84 \times 10^{19}$ сатоши, что соответствует 184 миллиардам биткойнов. Отрицательной суммы оплаты такой транзакции оказалось достаточно, чтобы программа на языке C++ приняла на веру, что эта сумма на самом деле положительная и равна 0,1 биткойна!

Эта уязвимость подробно описана в документе CVE-2010-5139 и устранена через так называемую "мягкую вилку" (т.е. нерадикальное изменение протокола) в версии Bitcoin Core 0.3.11. Упомянутая выше транзакция и созданные в ней биткойны были признаны недействительными задним числом в результате реорганизации блоков. Иными словами, блок, включавший транзакцию с превышением стоимости, а также все основанные на нем блоки были заменены.

Проверка подписи

Самой, вероятно, трудной частью проверки транзакции на достоверность является процесс проверки ее подписей. Как правило, у транзакции имеется хотя бы одна подпись на каждый ввод. Если же имеются выводы со многими подписями, которые могут быть израсходованы, то таких подписей может быть больше одной. Как пояснялось в главе 4, алгоритму цифровых подписей ECDSA требуются открытый ключ P, хеш подписи z и сама подпись (r,s). Если все эти исходные данные известны, то процесс верификации подписи существенно упрощается, как уже демонстрировалось на конкретных примерах кода в главе 3 и показано ниже.

```
>>> from ecc import S256Point, Signature
>>> sec = bytes.fromhex('0349fc4e631e3624a545de3f89f5d8684c7b\
8138bd94bdd531d2e213bf016b278a')
>>> der = bytes.fromhex('3045022100ed81ff192e75a3fd2304004dcadb\
746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb35d446a89\
d3f56100f4d7f67801c31967743a9c8e10615bed')
>>> z = 0x27e0c5994dec7824e56dec6b2fcb342eb7cdb0d0957\
c2fce9882f715e85d81a6
>>> point = S256Point.parse(sec)
>>> signature = Signature.parse(der)
>>> print(point.verify(z, signature))
True
```

Открытые ключи формата SEC и подписи формата DER находятся в стеке, когда выполняется такая операция, как `OP_CHECKSIG`, благодаря чему значительно упрощается получение открытого ключа и подписи (см. главу 6). А вот получить хеш подписи труднее. Наивный способ сделать это состоит в том, чтобы хешировать результат сериализации транзакции, как показано на рис. 7.1. Но, к сожалению, сделать это не удастся, поскольку подпись является частью сценария ScriptSig и не может сама подписаться.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
06b483045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf21320b0277457c98f02
207a986d955c6e0cb35d446a89d3f56100f4d7f67801c31967743a9c8e10615bed01210349fc4e631
e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278afeffffff02a135ef0100000000
1976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c3980000000000001976a9141c4bc
762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 7.1. Подпись находится в сценарии ScriptSig, выделенном светло-серым, начиная с байтов `6b48` *и заканчивая байтами* `278a`

Вместо этого можно модифицировать транзакцию перед тем, как ее подписывать. Это означает, что для *каждого* ввода вычисляется разный хеш подписи. Соответствующая процедура описана ниже.

Шаг 1. Опорожнение всех полей ScriptSig

Первый шаг состоит в том, чтобы опорожнить все поля ScriptSig при проверке подписи (рис. 7.2). Аналогичная процедура выполняется и для создания подписи, за исключением того, что поля ScriptSig, как правило, уже пусты.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
0000feffffff02a135ef0100000001976a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac
99c3980000000001976a9141c4bc762dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 7.2. Поле каждого ввода должно быть опорожнено (выделенное светло-серым поле теперь содержит байт `00`*)*

Следует, однако, иметь в виду, что в данном примере транзакции имеется единственный ввод, и поэтому опорожняется поле ScriptSig только этого ввода, хотя у транзакции может быть не один ввод. В таком случае должны быть опорожнены поля ScriptSig на всех вводах.

Шаг 2. Замена сценария ScriptSig из подписываемого ввода сценарием ScriptPubKey из вывода предыдущей транзакции

Как упоминалось ранее, каждый ввод указывает на вывод из предыдущей транзакции, где имеется поле ScriptPubKey. На рис. 7.3 еще раз приведена для напоминания блок-схема из главы 6, иллюстрирующая это положение.

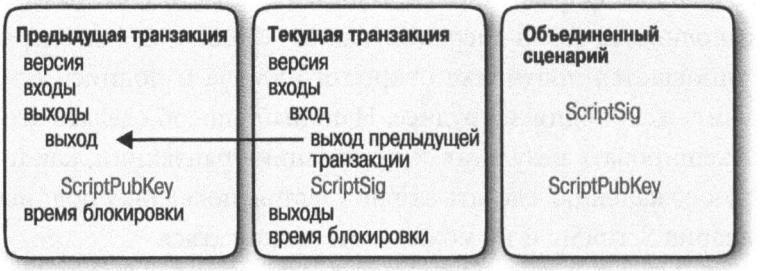

Рис. 7.3. Объединение сценариев ScriptPubKey и ScriptSig

В данном случае берется сценарий ScriptPubKey из вывода, на который указывает ввод из текущей транзакции, и ставится на место пустого поля сценария ScriptSig (рис. 7.4). Для этого, возможно, придется совершить поиск в блокчейне, но на практике подписывающему уже известен открытый ключ из сценария ScriptPubKey, поскольку на данном вводе у него имеется секретный ключ.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
01976a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88acffffffff02a135ef010000000019
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc76
2dd5423e332166702cb75f40df79fea1288ac19430600
```

Рис. 7.4. Замена поля ScriptSig, выделенного светло-серым, на одном из вводов текущей транзакции полем ScriptPubKey из ввода предыдущей транзакции

Шаг 3. Присоединение типа хеша

И на последнем шаге в конце присоединяется 4-байтовый тип хеша. Это делается для того, чтобы указать, что именно разрешает подпись. В частности, подпись может разрешить данному вводу присоединиться ко всем остальным вводам и выводам (`SIGHASH_ALL`), к конкретному выводу (`SIGHASH_SINGLE`) или же к любому выводу (`SIGHASH_NONE`). Две последние возможности осуществимы теоретически, но на практике почти все транзакции подписываются с типом хеша `SIGHASH_ALL`. Имеется также редко используемый тип хеша `SIGHASH_ANYONECANPAY`, который можно сочетать с любым из трех упомянутых выше типов хеша, но здесь он не рассматривается. Что же касается типа хеша `SIGHASH_ALL`, то в окончательно сформированной транзакции должно быть столько выводов, сколько было подписано, а иначе подпись на вводе будет недействительной.

Типу хеша SIGHASH_ALL соответствует целое число 1, которое должно быть закодировано 4 байтами в прямом порядке их следования. В итоге модификации транзакция будет выглядеть так, как показано на рис. 7.5.

```
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71bf8303c6a989c7d10000000
01976a914a802fc56c704ce87c42d7c92eb75e7896bdc41ae88acfeffffff02a135ef010000000019
76a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976a9141c4bc76
2dd5423e332166702cb75f40df79fea1288ac1943060001000000
```

*Рис. 7.5. Присоединение типа хеша (*SIGHASH_ALL*), закодированного 4 байтами* 01000000*, в самом конце*

Чтобы получить хеш подписи *z*, хеш-код hash256 этой модифицированной транзакции следует интерпретировать как целое число, представленное байтами в обратном порядке их следования. Ниже показано, каким образом хеш-код hash256 модифицированной транзакции преобразуется в хеш подписи *z* на языке Python.

```
>>> from helper import hash256
>>> modified_tx = bytes.fromhex('0100000001813f79011acb80925dfe69\
b3def355fe914bd1d96a3f5f71bf8303c6a989c7d1000000001976a914a802fc56\
c704ce87c42d7c92eb75e7896bdc41ae88acfeffffff02a135ef01000000001976\
a914bc3b654dca7e56b04dca18f2566cdaf02e8d9ada88ac99c39800000000001976\
a9141c4bc762dd5423e332166702cb75f40df79fea1288ac1943060001000000')
>>> h256 = hash256(modified_tx)
>>> z = int.from_bytes(h256, 'big')
>>> print(hex(z))
0x27e0c5994dec7824e56dec6b2fcb342eb7cdb0d0957c2fce9882f715e85d81a6
```

А теперь, когда имеется хеш подписи *z*, можно взять открытый ключ в формате SEC и подпись в формате DER из поля ScriptSig, чтобы произвести верификацию подписи следующим образом:

```
>>> from ecc import S256Point, Signature
>>> sec = bytes.fromhex('0349fc4e631e3624a545de3f89f5d8684\
c7b8138bd94bdd531d2e213bf016b278a')
>>> der = bytes.fromhex('3045022100ed81ff192e75a3fd2304004dc\
adb746fa5e24c5031ccfcf21320b0277457c98f02207a986d955c6e0cb35d\
446a89d3f56100f4d7f67801c31967743a9c8e10615bed')
>>> z = 0x27e0c5994dec7824e56dec6b2fcb342eb7cdb0d0957\
c2fce9882f715e85d81a6
>>> point = S256Point.parse(sec)
>>> signature = Signature.parse(der)
>>> point.verify(z, signature)
True
```

Описанный выше процесс проверки транзакции на достоверность можно было бы запрограммировать в отдельном методе из класса `Tx`. Правда, интерпретатор Script уже обладает способностью производить верификацию подписей (см. главу 6), поэтому нам остается лишь слепить все вместе. Для этого необходимо передать хеш подписи *z* методу `evaluate()` и объединить сценарии ScriptSig и ScriptPubKey.

Квадратичное хеширование

Алгоритм хеширования подписей неэффективен и нерационален. Как утверждает так называемая *проблема квадратичного хеширования*, время, требующееся для вычисления хешей подписей, возрастает в квадрате от количества вводов, имеющихся в транзакции. В частности, увеличивается не только количество операций хеширования по алгоритму hash256 для вычисления хеша подписей *z* по каждому вводу, но и длина транзакции. А это замедляет выполнение операции хеширования по алгоритму hash256, поскольку весь хеш подписей приходится вычислять заново для каждого ввода.

Это стало особенно очевидно на примере самой крупной транзакции, добытой до сих пор:

`bb41a757f405890fb0f5856228e23b715702d714d59bf2b1feb70d8b2b4e3e08`

У этой транзакции было 5569 вводов и 1 вывод, а для ее проверки на достоверность пришлось задействовать многих добытчиков криптовалюты, поскольку вычисление хешей подписей для этой транзакции обошлось бы очень дорого. В протоколе Segwit (см. главу 13) данная проблема решается иным способом вычисления хеша подписи, как описывается в документации к протоколу BIP0143.

Упражнение 1

Напишите метод `sig_hash()` для класса `Tx`, чтобы реализовать в нем вычисление хеша подписи.

Упражнение 2

Напишите метод `verify_input()` для класса `Tx`, чтобы реализовать в нем проверку ввода транзакции на достоверность. Для этого вам придется воспользоваться методами `TxIn.script_pubkey()`, `Tx.sig_hash()` и `Script.evaluate()`.

Верификация всей транзакции

А теперь, когда можно проверить ввод транзакции, решить задачу верификации всей транзакции в целом не составит большого труда, как показано ниже.

```
class Tx:
...
  def verify(self):
    '''Произвести верификацию данной транзакции'''
    if self.fee() < 0:    ❶
      return False
    for i in range(len(self.tx_ins)):
      if not self.verify_input(i):    ❷
        return False
    return True
```

❶ Здесь проверяется, не создаются ли биткойны в данной транзакции.

❷ Здесь проверяется, имеется ли у каждого ввода правильное поле ScriptSig.

Следует, однако, иметь в виду, что в полном узле поверяется не только попытка создать биткойны в транзакции, но и многое другое. В частности, двойное расходование и соблюдение других согласованных правил, не рассматриваемых в этой главе (максимальное количество операций подписи на блок, размер поля ScriptSig и т.д.). Впрочем, это совсем неплохо для рассматриваемой здесь библиотеки.

Создание транзакции

Приведенный выше код для верификации транзакций окажет нам немалую услугу в создании транзакций. Таким образом, мы сможем создавать транзакции, вписывающиеся в процесс верификации. В тех транзакциях, которые нам предстоит здесь создать, требуется, чтобы сумма вводов была больше или равна сумме выводов и чтобы сценарий, объединенный из исходных сценариев ScriptSig и ScriptPubKey, был достоверным.

Чтобы создать транзакцию, нам потребуется хотя бы один из полученных нами выводов. Это означает, что нам потребуется вывод из множества UTXO, где можно разблокировать открытый ключ из сценария ScriptPubKey. И нам в основном потребуется один или несколько секретных ключей, соответствующих открытым ключам, хешированным в сценарии ScriptPubKey. Остальная часть этой главы посвящена созданию транзакции, вводы которой заблокированы открытыми ключами из сценария ScriptPubKey для p2pkh.

Построение транзакции

Для построения транзакции придется ответить на следующие основные вопросы.

1. Куда должны быть направлены биткойны?
2. Какие неизрасходованные выводы можно израсходовать?
3. Насколько быстро требуется доставить данную транзакцию в блокчейн?

Для рассматриваемого здесь примера мы воспользуемся сетью testnet, хотя его нетрудно распространить и на сеть mainnet. Первый из перечисленных выше вопросов касается суммы оплаты и места ее назначения. Оплату можно произвести по одному или нескольким адресам. В данном примере будет произведена оплата на сумму 0,1 tBTC (т.е. биткойнами в сети testnet) по адресу `mnrVtF8DWjMu839VW3rBfgYaAfKk8983Xf`.

Второй вопрос касается содержимого нашего кошелька, т.е. чем мы располагаем, чтобы платить. В данном примере у нас имеется вывод транзакции, обозначенный идентификатором и индексом, как показано ниже.

```
0d6fe5213c0b3291f208cba8bfb59b7476dffacc4e5cb66f6eb20a080843a299:13
```

Просматривая этот вывод в обозревателе блоков в сети testnet (рис. 7.6), мы обнаружим, что его стоимость равна 0,44 tBTC.

Эта стоимость превышает 0,1 tBTC, а потому остаток нам придется отправить самим себе. И хотя повторное использование адресов, в общем, считается ненадлежащей нормой соблюдения конфиденциальности и безопасности, мы все же отправим оставшиеся биткойны обратно по адресу `mzx5YhAH9kNHtcN481u6WkjeHjYtVeKVh2`, чтобы упростить построение транзакции.

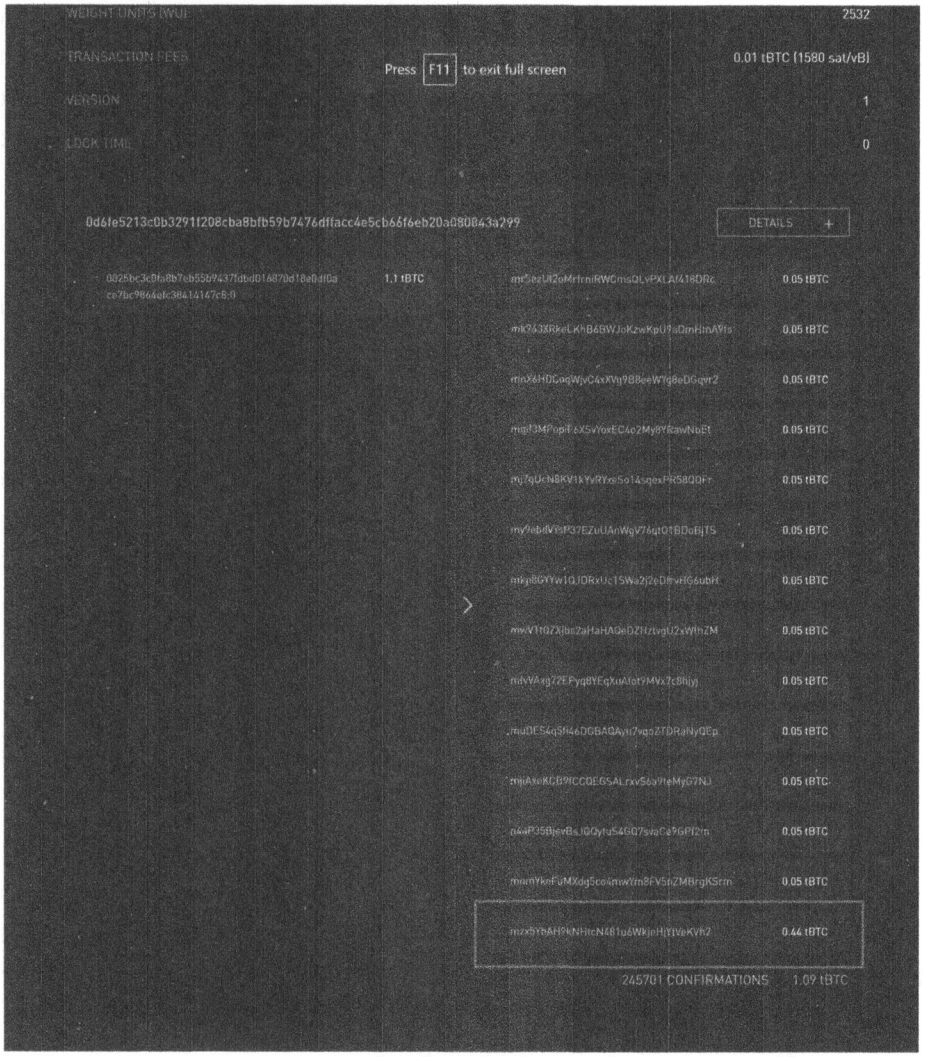

Рис. 7.6. Неизрасходованный вывод UTXO, который мы собираемся израсходовать

Почему повторно пользоваться адресами не рекомендуется

В главе 6 пояснялось, что сценарий p2pk уступает сценарию p2pkh отчасти потому, что он был защищен только алгоритмом ECDSA. А с другой стороны, сценарий p2pkh дополнительно защищен алгоритмами sha256 и ripemd160. Но поскольку блокчейн общедоступен, израсходовав однажды биткойны из сценария

ScriptPubKey, соответствующего нашему адресу, мы обнаружим, что наш открытый ключ является частью сценария ScriptSig. А раз так, то алгоритмы sha256 и ripemd160 не смогут нас больше защитить, поскольку атакующему злоумышленнику известен открытый ключ, не требующий разгадки.

На момент написания этой книги задача дискретного логарифмирования все еще обеспечивала необходимую защиту, хотя она может быть рано или поздно решена. Но из соображений безопасности очень важно понимать, что именно нас защищает.

Еще одна причина не пользоваться адресами повторно связана с конфиденциальностью. Наличие одного адреса для всех совершаемых нами транзакций означает, что другие могут связать их вместе. Так, если мы приобрели нечто личное (например, лекарства от какой-нибудь болезни, о которой не следует знать посторонним) и израсходовали другой вывод по тому же самому открытому ключу из сценария ScriptPubKey на благотворительность, то благотворительная организация может выявить, что мы имели дело с поставщиком лекарств, а тот — с ней. Утечки конфиденциальности со временем становятся брешами в защите.

И наконец, третий вопрос связан с платой за транзакции. Так, если нам требуется быстрее доставить транзакцию в блокчейн, за это нам придется заплатить больше. А если мы готовы подождать, то можем заплатить меньше. В рассматриваемом здесь примере плата за транзакцию составляет 0,01 tBTC.

Оценка платы за транзакцию

Оценка платы за транзакцию, иначе называемой комиссией, производится побайтно. Так, если транзакция состоит из 600 байтов, плата за нее окажется в два раза больше, чем за транзакцию длиной 300 байтов. Дело в том, что место в блоке ограничено, и чем крупнее транзакции, тем больше им требуется места. С появлением протокола Segwit (см. главу 13) такой порядок расчета платы за транзакцию немного изменился, хотя общий принцип остается прежним. Нам придется заплатить побайтно в такой мере, чтобы добытчики криптовалюты были заинтересованы включить нашу транзакцию в цепочку блоков как можно скорее.

Если блоки еще не заполнены, то почти любой суммы свыше стандартного предела пересылки (1 сатоши/байт) оказывается достаточно, чтобы включить транзакцию в цепочку блоков. А если блоки уже заполнены, то оценить плату за транзакцию непросто. Имеются разные способы оценки платы за транзакции, в том числе следующие.

- Анализ различных уровней оплаты и оценка вероятности включения транзакции в цепочку блоков на основании прошлых блоков и пулов памяти на данный момент.
- Анализ текущего пула памяти и прибавление платы, приблизительно соответствующей достаточной материальной заинтересованности.
- Довольствование фиксированной платой.

Во многих программных кошельках применяются разные стратегии платы за транзакции. В этой области проводятся активные исследования.

Составление транзакции

Итак, мы собираемся создать новую транзакцию с одним вводом и двумя выводами. Но прежде рассмотрим ряд других средств, которые могут нам в этом пригодиться.

В частности, нам необходимо каким-то образом преобразовать адрес в 20-байтовый хеш-код. Это операция, противоположная кодированию адреса, и поэтому присвоим выполняющей ее функции имя `decode_base58`. Ниже показано, каким образом она определяется непосредственно в коде.

```
def decode_base58(s):
  num = 0
  for c in s:     ❶
    num *= 58
    num += BASE58_ALPHABET.index(c)
  combined = num.to_bytes(25, byteorder='big')     ❷
  checksum = combined[-4:]
  if hash256(combined[:-4])[:4] != checksum:
    raise ValueError('bad address: {} {}'
          .format(checksum, hash256(combined[:-4])[:4]))
  return combined[1:-4]     ❸
```

❶ Здесь выясняется, в какое именно число кодируется данный адрес в кодировке Base58.

❷ Получив число, можно преобразовать его в байты, следующие в обратном порядке.

❸ Первый байт обозначает сетевой префикс, последние четыре байта — контрольную сумму, а находящиеся между ними 20 байтов — 20-байтовый хеш-код hash160.

Кроме того, нам требуется каким-то образом преобразовать 20-байтовый хеш-код в сценарий ScriptPubKey. Эта операция реализуется в функции, называемой `p2pkh_script`, поскольку в ней хеш-код hash160 преобразуется в сценарий ScriptPubKey для p2pkh, как показано ниже.

```
def p2pkh_script(h160):
  '''Принимает хеш-код hash160 и возвращает сценарий
     ScriptPubKey для p2pkh'''
  return Script([0x76, 0xa9, h160, 0x88, 0xac])
```

Следует иметь в виду, что байт `0x76` обозначает код операции `OP_DUP`, байт `0xa9` — код операции `OP_HASH160`, байт `h160` — 20-байтовый элемент, байт `0x88` — код операции `OP_EQUALVERIFY`, а байт `0xac` — код операции `OP_CHECKSIG`. По существу, это набор команд из сценария ScriptPubKey для p2pkh, упоминавшегося в главе 6.

Имея в своем распоряжении упомянутые выше средства, мы можем перейти непосредственно к созданию транзакции следующим образом:

```
>>> from helper import decode_base58, SIGHASH_ALL
>>> from script import p2pkh_script, Script
>>> from tx import TxIn, TxOut, Tx
>>> prev_tx = bytes.fromhex('0d6fe5213c0b3291f208cba8bfb59b7476\
dffacc4e5cb66f6eb20a080843a299')
>>> prev_index = 13
>>> tx_in = TxIn(prev_tx, prev_index)
>>> tx_outs = []
>>> change_amount = int(0.33*100000000)     ❶
>>> change_h160 = decode_base58('mzx5YhAH9kNHtcN481u6WkjeHjYtVeKVh2')
>>> change_script = p2pkh_script(change_h160)
>>> change_output = TxOut(amount=change_amount,
                          script_pubkey=change_script)
>>> target_amount = int(0.1*100000000)      ❶
>>> target_h160 = decode_base58('mnrVtF8DWjMu839VW3rBfgYaAfKk8983Xf')
>>> target_script = p2pkh_script(target_h160)
>>> target_output = TxOut(amount=target_amount,
                          script_pubkey=target_script)
```

```
>>> tx_obj = Tx(1, [tx_in], [change_output, target_output],
                0, True)     ❷
>>> print(tx_obj)
tx: cd30a8da777d28ef0e61efe68a9f7c559c1d3e5bcd7b265c850ccb4068598d11
c850ccb4068598d11
version: 1
tx_ins:
0d6fe5213c0b3291f208cba8bfb59b7476dffacc4e5cb66f6eb20a080843a299:13
tx_outs:
33000000:OP_DUP OP_HASH160 d52ad7ca9b3d096a38e752c2018e6fbc40cdf26f \
OP_EQUALVERIFY OP_CHECKSIG
10000000:OP_DUP OP_HASH160 507b27411ccf7f16f10297de6cef3f291623eddf \
OP_EQUALVERIFY OP_CHECKSIG
locktime: 0
```

❶ Сумма должна быть указана в сатоши. Принимая во внимание, что на 1 биткойн равняется 100000000 сатоши, эту сумму необходимо умножить на 100000000 и привести к целочисленному значению.

❷ Здесь с помощью аргумента `testnet=True` указывается, в какой именно сети следует производить поиск.

Итак, мы создали конкретную транзакцию, но каждое поле ScriptSig в этой транзакции в настоящий момент пусто. Его заполнению и будет посвящен следующий раздел.

Подписание транзакции

И хотя подписание транзакции может оказаться совсем нелегким делом, ранее в этой главе мы уже выяснили, как получить хеш подписи *z*. Так, если у нас имеется секретный ключ, открытый ключ которого преобразуется по алгоритму hash160 в 20-байтовый хеш по сценарию ScriptPubKey, мы можем подписать хеш *z* и получить подпись в формате DER, как показано ниже.

```
>>> from ecc import PrivateKey
>>> from helper import SIGHASH_ALL
>>> z = transaction.sig_hash(0)     ❶
>>> private_key = PrivateKey(secret=8675309)
>>> der = private_key.sign(z).der()
>>> sig = der + SIGHASH_ALL.to_bytes(1, 'big')     ❷
>>> sec = private_key.point.sec()
>>> script_sig = Script([sig, sec])     ❸
>>> transaction.tx_ins[0].script_sig = script_sig     ❹
>>> print(transaction.serialize().hex())
0100000001813f79011acb80925dfe69b3def355fe914bd1d96a3f5f71\
bf8303c6a989c7d1000000006a47304402207db2402a3311a3b845b038885\
```

```
e3dd889c08126a8570f26a844e3e4049c482a11022010178cdca4129eacbe\
ab7c44648bf5ac1f9cac217cd609d216ec2ebc8d242c0a012103935581e52\
cd2f484fe8ed83af7a3097005b2f9c60bff71d35bd795f54b67ffffff02\
a135c354ef01000000001976a914bc3b654dca7e56b04dca18f2566cdaf02
\e8d9ada88ac99c39800000000001976a9141c4bc762dd5423e332166702\
cb75f40df79fea1288ac19430600
```

❶ Здесь требуется подписать лишь первый и единственный ввод. А для подписания нескольких вводов каждый из них придется подписывать подходящим секретным ключом.

❷ Подпись фактически состоит из самой подписи в формате DER и типа хеша (в данном случае — `SIGHASH_ALL`).

❸ Сценарий ScriptSig для p2pkh состоит ровно из двух элементов, как пояснялось в главе 6: подписи и открытого ключа в формате SEC.

❹ В данном случае требуется подписать лишь один ввод, но если бы вводов было больше, процесс создания сценария ScriptSig пришлось бы выполнить для каждого ввода.

Упражнение 3

Напишите метод `sign_input()` для класса `Tx`, чтобы реализовать в нем подписание вводов транзакции.

Создание собственных транзакций в сети testnet

Чтобы создать свою транзакцию, необходимо приобрести немного монет, а для этого потребуется адрес. Если вы выполнили последнее упражнение из главы 4, у вас должны быть свой адрес в сети testnet и секретный ключ. Если же вы не помните, ниже показано, как это делается.

```
>>> from ecc import PrivateKey
>>> from helper import hash256, little_endian_to_int
>>> secret = little_endian_to_int(hash256(b'Jimmy Song secret'))     ❶
>>> private_key = PrivateKey(secret)
>>> print(private_key.point.address(testnet=True))
mn81594PzKZa9K3Jyy1ushpuEzrnTnxhVg
```

❶ Непременно используйте другую строку вместо `'Jimmy Song secret'`.

Как только у вас появится свой адрес, вы сможете приобрести немного монет из одного из тех биткойновых кранов в сети testnet, где бесплатно предоставляются монеты для тестирования. Чтобы найти такой кран в

сети testnet, введите критерий *testnet bitcoin faucet* или *биткойновый кран в testnet* в поисковом механизме Google или выберите один из них из списка в разделе Faucets (Краны) на вики-странице, доступной по адресу `https://en.bitcoin.it/wiki/Testnet#Faucets`. Введите свой новый адрес в сети testnet в любом из выбранных вами биткойновых кранов, чтобы приобрести немного бесплатных биткойнов для тестирования.

Получив немного монет, попробуйте израсходовать их, используя рассматриваемую здесь библиотеку. Это хорошая школа для начинающих разработчиков биткойна, поэтому уделите немного времени выполнению приведенных ниже упражнений.

Упражнение 4

Создайте транзакцию для отправки 60% суммы из одного неизрасходованного вывода UTXO по адресу `mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv` в сети testnet. Оставшаяся сумма за вычетом платы за транзакцию должна вернуться обратно по вашему измененному адресу. Это должна быть транзакция с одним вводом и двумя выводами. Свою транзакцию вы можете переслать по адресу `https://live.blockcypher.com/btc/pushtx`.

Упражнение 5

Это упражнение посложнее. Приобретите немного монет из биткойнового крана в сети testnet и создайте транзакцию с двумя вводами и одним выводом. Один из вводов должен быть из биткойнового крана, другой — из предыдущего упражнения, а выводом может служить ваш собственный адрес. Свою транзакцию вы можете переслать по адресу `https://live.blockcypher.com/btc/pushtx`.

Заключение

Проработав материал этой главы, вы научились успешно создавать и проверять транзакции на достоверность в блокчейне, опробовав их в сети testnet. И это немалое достижение, которым стоит гордиться!

Написанный нами до сих пор код позволяет выполнять сценарии p2pkh и p2pk. А в следующей главе мы обратимся к более развитому умному контракту по сценарию p2sh.

ГЛАВА 8

Оплата по хешу сценария

До сих пор в этой книге рассматривались транзакции с единственным ключом или транзакции с одним секретным ключом на каждый ввод. А что делать, если требуется нечто более сложное? Например, компании, вложившей 100 миллионов долларов США в биткойн, вряд ли подойдет единственный секретный ключ для блокировки ее денежных средств. Ведь если этот ключ будет потерян или украден, все ее денежные средства будут утрачены. Что же тогда предпринять, чтобы снизить риск от наличия этой единственной уязвимой точки отказа?

Решение состоит в применении *мультиподписей*. Такая возможность была встроена в биткойн с самого начала, но поначалу она была слишком громоздкой и потому не применялась. Как поясняется далее в этой главе, Сатоши Накомото не проверял мультиподпись, поскольку ей была присуща ошибка смещения на единицу (см. далее раздел "Ошибка смещения на единицу в операции OP_CHECKMULTISIG"). Эта ошибка так и осталась в протоколе потому, что для ее исправления потребовалась бы жесткая вилка, т.е. радикальное изменение в самом протоколе.

Несколько секретных ключей к одному составному открытому ключу

Один секретный ключ можно "разбить" на несколько секретных ключей и воспользоваться интерактивным способом для составления подписей, не восстанавливая сам секретный ключ, хотя такая норма не является общепринятой. Подписи Шнорра упрощают процесс составления подписей и поэтому, вероятнее всего, найдут более широкое распространение в будущем.

Простая мультиподпись

Простая мультиподпись была первой попыткой создать выводы транзакции, требующие подписей нескольких сторон. Ее замысел состоял в том, чтобы перейти от единственной точки отказа к чему-то более взломоустойчивому. Чтобы понять суть простой мультиподписи, необходимо сначала уяснить операцию `OP_CHECKMULTISIG`. Как пояснялось в главе 6, в языке Script имеются самые разные коды операций, в том числе код `0xae` операции `OP_CHECKMULTISIG`. Эта операция извлекает немало элементов из стека и возвращает требующееся количество подписей, действительных для ввода транзакции.

Вывод транзакции называется "простой мультиподписью" из-за своего длинного поля ScriptPubKey. На рис. 8.1 показано, как выглядит поле ScriptPubKey для мультиподписи типа "1 из 2".

```
514104fcf07bb1222f7925f2b7cc15183a40443c578e62ea17100aa3b44b
a66905c95d4980aec4cd2f6eb426d1b1ec45d76724f26901099416b9265b
76ba67c8b0b73d210202be80a0ca69c0e000b97d507f45b98c49f58fec66
50b64ff70e6ffccc3e6d0052ae

- 51 - OP_1
- 41 - Длина <открытого ключа 1>
- 04fc...3d - <открытый ключ 1>
- 21 - Длина <открытого ключа 2>
- 0202...00 - <открытый ключ 2>
- 52 - OP_2
- ae - OP_CHECKMULTISIG
```

Рис. 8.1. Поле ScriptPubKey для простой мультиподписи

И хотя это одно из самых коротких среди всех полей ScriptPubKey простых мультиподписей, уже видно, что оно довольно длинное. Если поле ScriptPubKey для p2pkh составляет лишь 25 байтов, то поле ScriptPubKey для столь простой мультиподписи — 101 байт, хотя его длину можно немного сократить, используя сжатый формат SEC. А ведь это всего лишь мультиподпись типа "1 из 2"! На рис. 8.2 показано, как выглядит поле ScriptSig для простой мультиподписи.

Для данной мультиподписи типа "1 из 2" требуется лишь одна подпись, и поэтому она оказывается относительно короткой. А вот для мультиподписи типа "5 из 7" потребуется 5 подписей формата DER, и поэтому она окажется намного более длинной (около 360 байтов). На рис. 8.3 показано, каким образом объединяются сценарии ScriptSig и ScriptPubKey для мультиподписи.

```
00483045022100e222a0a6816475d85ad28fbeb66e97c931081076dc9655
da3afc6c1d81b43f9802204681f9ea9d52a31c9c47cf78b71410ecae6188
d7c31495f5f1adfe0df5864a7401

- 00 - OP_0
- 48 - Длина <подписи 1>
- 3045...01 - <подпись 1>
```

Рис. 8.2. Поле ScriptSig для простой мультиподписи

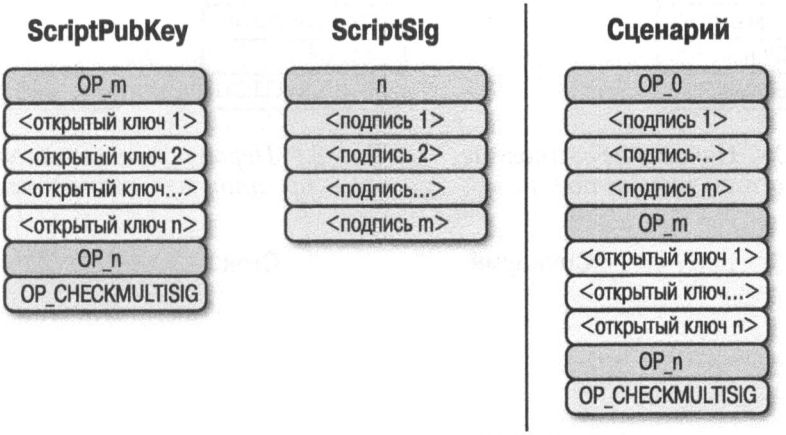

Рис. 8.3. Объединенный сценарий для мультиподписи

Данный пример обобщен для демонстрации того, как будет выглядеть простая мультиподпись типа "*m* из *n*", где *m* и *n* могут быть любыми числами в пределах от 1 до 20 включительно. И хотя числовые коды операций доходят лишь до `OP_16`, для числовых значений от 17 до 20 потребуется код операции `0112`, чтобы разместить в стеке, например, число 18. Начальное состояние простой мультиподписи выглядит так, как показано на рис. 8.4.

Операция `OP_0` размещает в стеке число 0 (рис. 8.5).

Подписи являются элементами, и поэтому они будут размещаться непосредственно в стеке (рис. 8.6).

Операция `OP_m` размещает в стеке число *m*, а также открытые ключи. А операция `OP_n` размещает в стеке число *n* (рис. 8.7).

Рис. 8.4. Начальное состояние простой мультиподписи

Рис. 8.5. Первая стадия выполнения простой мультиподписи

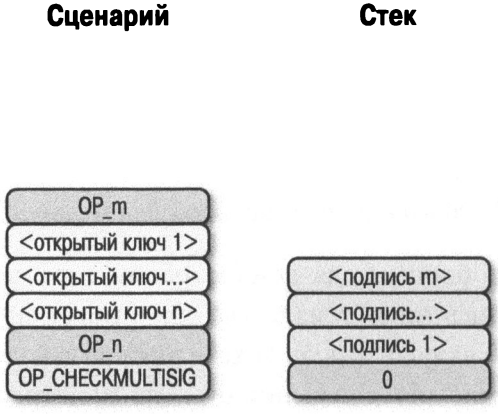

Рис. 8.6. Вторая стадия выполнения простой мультиподписи

Рис. 8.7. Третья стадия выполнения простой мультиподписи

На данной стадии операция OP_CHECKMULTISIG извлекает $m + n + 3$ элементов (см. далее раздел "Ошибка смещения на единицу в операции OP_CHECKMULTISIG") и размещает 1 в стеке, если m подписей действительны для m отдельных открытых ключей из списка n открытых ключей, а иначе в стеке размещается 0. Если допустить, что подписи действительны, в стеке останется только 1, подтверждающая достоверность объединенного сценария (рис. 8.8).

Рис. 8.8. Конечное состояние простой мультиподписи

Ошибка смещения на единицу в операции OP_CHECKMULTISIG

Элементы, извлекаемые из стека в операции OP_CHECKMULTISIG, предположительно состоят из числа m, m разных подписей, числа n и n разных открытых ключей. Количество извлекаемых элементов должно быть равно 2 (самих чисел m и n) + m (подписей) + n (открытых ключей). Но, к сожалению, данная операция извлекает из стека на один элемент больше, чем $m + n + 2$ предполагаемых элементов. На самом деле операция OP_CHECKMULTISIG извлекает из стека $m + n + 3$ элементов, и, таким образом, прибавляется лишний элемент (в данном случае — OP_0), что не приводит к неудачному завершению данной операции.

Данная операция ничего не делает с лишним элементом, который может быть каким угодно. Но в качестве меры борьбы с податливостью в большинстве узлов сети биткойна транзакция не будет пересылаться, если только лишним окажется элемент OP_0. Следует, однако, иметь в виду, что если бы имелось $m + n + 2$ элементов, операция OP_CHECKMULTISIG завершилась бы неудачно, поскольку извлекаемых элементов оказалось бы недостаточно. Это привело бы к неудачному завершению объединенного сценария, а вся транзакция в целом оказалась бы недостоверной.

Программная реализация операции OP_CHECKMULTISIG

В простой мультиподписи типа "*m* из *n*" на вершине стека находится элемент *n*, а далее следуют *n* открытых ключей, число *m*, *m* подписей и, наконец, лишнего элемента из-за ошибки смещения на единицу. Исходный код, реализующий операцию OP_CHECKMULTISIG в исходном файле op.py, почти написан и приведен ниже.

```
def op_checkmultisig(stack, z):
  if len(stack) < 1:
    return False
  n = decode_num(stack.pop())
  if len(stack) < n + 1:
    return False
  sec_pubkeys = []
  for _ in range(n):
    sec_pubkeys.append(stack.pop())
  m = decode_num(stack.pop())
  if len(stack) < m + 1:
    return False
  der_signatures = []
  for _ in range(m):
    der_signatures.append(stack.pop()[:-1])     ❶
  stack.pop()     ❷
  try:
    raise NotImplementedError     ❸
  except (ValueError, SyntaxError):
    return False
return True
```

❶ Предполагается, что каждая подпись формата DER делается с типом хеша SIGHASH_ALL.

❷ Здесь учитывается ошибка смещения на единицу, для чего из вершины стека извлекается элемент, но с ним ничего, по существу, не делается.

❸ Эту часть исходного кода придется дописать в следующем упражнении.

Упражнение 1

Напишите окончательный вариант функции op_checkmultisig() из исходного файла op.py.

Недостатки простой мультиподписи

Простая мультиподпись неказиста, хотя и вполне работоспособна. Она исключает единственную точку отказа, требуя *m* из *n* подписей для разблокировки неизрасходованного вывода UTXO. Имеется немало возможностей сделать выводы с мультиподписью, особенно в деловой сфере. Но простая мультиподпись страдает следующими недостатками.

1. В поле ScriptPubKey для простой мультиподписи находятся самые разные открытые ключи, и поэтому оно такое длинное. В отличие от полей для p2pkh и даже p2pk, передать содержимое такого поля, используя голосовые или текстовые сообщения, не так-то просто.
2. В связи с тем, что вывод оказывается довольно длинным, в 5–20 раз превышая по длине обычный вывод для p2pkh, его обработка требует больше ресурсов от программного обеспечения узла. В узлах отслеживается множество UTXO, а отслеживание длинного поля ScriptPubKey обходится дороже, как, впрочем, и хранение крупного вывода в запоминающем устройстве с быстрой выборкой (например, в оперативной памяти компьютера).
3. Простой мультиподписью можно злоупотребить, что и происходило на практике из-за того, что поле ScriptPubKey слишком длинное. Вся техническая документация, первоначально составленная Сатоши Накамото в формате PDF, закодирована в следующей транзакции из блока 230009:

   `54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713`

Создатель этой транзакции разделил техническую документацию в формате PDF на фрагменты по 64 байта, которые затем превратились в недостоверные открытые ключи. Эта техническая документация была закодирована в 947 выводов для мультиподписи типа "1 из 3". И хотя эти выводы не расходуются, они все же должны быть проиндексированы во множествах UTXO полных узлов. За это с каждого полного узла взимается плата, что и считается злоупотреблением. Для преодоления упомянутых выше недостатков и была придумана оплата по хешу сценария (p2sh).

Оплата по хешу сценария

Оплата по хешу сценария (p2sh) служит общим решением проблемы длинных адресов и полей ScriptPubKey. И хотя подобным способом могут быть

легко созданы более сложные сценарии ScriptPubKey, чем в простой мультиподписи, им, тем не менее, присущи такие же недостатки, как и у простой мультиподписи.

Решение, реализуемое в p2sh, состоит в том, чтобы взять хеш некоторых команд Script и выявить в дальнейшем команды Script нахождения прообраза. Оплата по хешу сценария была внедрена в 2011 году и вызвала немало противоречий. Несмотря на многие предложения, механизм p2sh страдает недостатками, но все же вполне работоспособен.

В p2sh выполняется особое правило лишь в том случае, если встречается последовательность команд, приведенная на рис. 8.9.

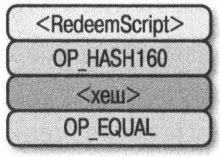

Рис. 8.9. Последовательность команд для оплаты по хешу сценария (p2sh), выполняющая особое правило

Если эта последовательность команд завершается 1 в стеке, то сценарий погашения (верхний элемент RedeemScript на рис. 8.9) подвергается синтаксическому анализу и затем вводится в набор команд Script. Эта особая последовательность команд была внедрена в протокол BIP0016, который реализуется программным обеспечением биткойна, проверяющим данную последовательность команд. Сценарий погашения RedeemScript не вводит новые команды Script для обработки, если только не встретится *именно такая* последовательность команд, завершающаяся 1.

Если это звучит нескладно, то так оно и есть. Но прежде чем дойти до сути, рассмотрим поточнее, как это проявляется. Допустим, что имеется сценарий ScriptPubKey для мультиподписи типа "2 из 2", как показано на рис. 8.10.

Это сценарий ScriptPubKey для простой мультиподписи. Нам требуется преобразовать его в сценарий p2sh, чтобы взять хеш этого сценария и сохранить данный сценарий до того момента, когда потребуется его погасить. Этот сценарий называется RedeemScript потому, что он обнаруживается только во время погашения. Хеш сценария RedeemScript размещается в поле сценария ScriptPubKey (рис. 8.11).

```
5221022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff129
5d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec45
3e1ffac7fbdbd4bb7152ae

- 52 - OP_2
- 21 - Длина <открытого ключа 1>
- 02...db70 - <открытый ключ 1>
- 21 - Длина <открытого ключа 2>
- 03...bb71 - <открытый ключ 2>
- 52 - OP_2
- ae - OP_CHECKMULTISIG
```

Рис. 8.10. Сценарий погашения RedeemScript для оплаты по хешу сценария (p2sh)

```
a91474d691da1574e6b3c192ecfb52cc8984ee7b6c5687

- a9 - OPHASH160
- 14 - Длина <хеша>
- 74d6...56 - <хеш>
- 87 - OP_EQUAL
```

Рис. 8.11. Сценарий ScriptPubKey для оплаты по хешу сценария (p2sh)

Здесь свертка хеша является хеш-кодом hash160 сценария RedeemScript или тем, что прежде было сценарием ScriptPubKey. Мы блокируем денежные средства в *хеше* сценария RedeemScript, который должен быть выявлен во время разблокировки.

Создание сценария ScriptSig для p2sh включает в себя не только выявление, но и разблокировку сценария RedeemScript. И здесь может возникнуть вопрос: а где хранится сценарий RedeemScript? Он не появится в блокчейне до тех пор, пока не настанет момент погашения, и поэтому он должен быть непременно сохранен создателем адреса p2sh. Если же сценарий RedeemScript потеряется и его нельзя будет восстановить, то будут утрачены и денежные средства, поэтому очень важно следить за ним!

Хранить сценарий RedeemScript очень важно!

Если вы получите адрес p2sh, непременно сохраните и сделайте резервную копию сценария RedeemScript! А еще лучше постарайтесь упростить его восстановление!

Сценарий ScriptSig для мультиподписи типа “2 из 2” выглядит так, как показано на рис. 8.12.

```
0048304502210 0dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e022
05a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a8993701483045022100da
6bee3c93766232079a01639d07fa8695987497 29ae323eab8eef53577d611b02207bef15429dcadce
2121ea07f233115c6f09034c0be68db99980b9a6c5e7540220147 5221022626e955ea6ea6d98850c9
94f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9feed096bed804
5c8b98bec453e1ffac7fbdbd4bb7152ae

- 00 - OP_0
- 48 - Длина <подписи 1>
- 3045...3701 - <подпись 1>
- 48 - Длина <подписи 2>
- 3045...2201 - <подпись 2>
- 47 - Length of <RedeemScript>
- 5221...ae - <RedeemScript>
```

Рис. 8.12. Сценарий ScriptSig для оплаты по хешу сценария (p2sh)

В итоге получается объединенный сценарий, как показано на рис. 8.13.

Рис. 8.13. Объединенный сценарий для p2sh

Как и прежде, в этом сценарии присутствует операция `OP_0` из-за ошибка смещения на единицу в операции `OP_CHECKMULTISIG`. Ключом к пониманию принципа действия p2sh служит точное выполнение последовательности команд, приведенной на рис. 8.14.

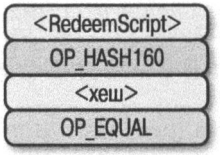

Рис. 8.14. Последовательность команд для p2sh, выполняющая особое правило

Если после выполнения этой последовательности команд в стеке останется 1, сценарий RedeemScript будет введен в набор команд Script. Иными

словами, если обнаружится сценарий RedeemScript с таким же самым хеш-кодом hash160, как и в сценарии ScriptPubKey, он будет действовать вместо сценария ScriptPubKey. Таким образом, сценарий, блокирующий денежные средства, хешируется и полученный хеш вводится в блокчейн вместо самого сценария. Именно поэтому такой сценарий называется оплатой по *хешу* сценария.

А теперь рассмотрим принцип его действия. Начинается сценарий p2sh с последовательности команд Script, приведенных на рис. 8.15.

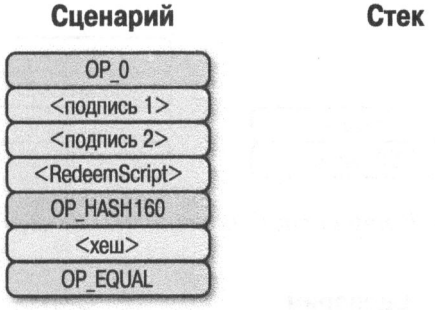

Рис. 8.15. Начальное состояние сценария p2sh

Операция `OP_0` размещает в стеке число 0, после чего в нем непосредственно размещаются две подписи и сценарий погашения RedeemScript, как показано на рис. 8.16.

Рис. 8.16. Первая стадия выполнения сценария p2sh

Операция `OP_HASH160` хеширует сценарий RedeemScript, в результате чего содержимое стека становится таким, как показано на рис. 8.17.

Далее 20-байтовый хеш-код размещается в стеке (рис. 8.18).

И наконец, операция `OP_EQUAL` сравнивает два верхних элемента в стеке. Если программное обеспечение, проверяющее такую транзакцию, было

разработано до появления протокола BIP0016, то в конечном счете будет получен результат, приведенный на рис. 8.19. На этом вычисление данного сценария в узлах, действующих по протоколу, предшествующему BIP0016, завершается, а результат оказывается достоверным, если сравниваемые хеши равны.

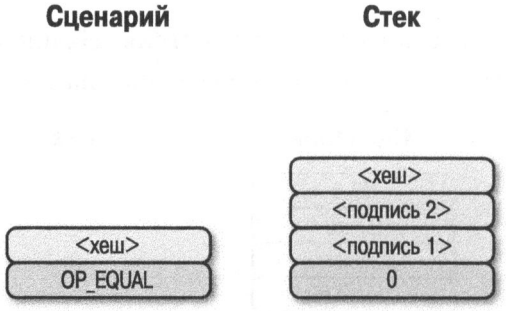

Рис. 8.17. Вторая стадия выполнения сценария p2sh

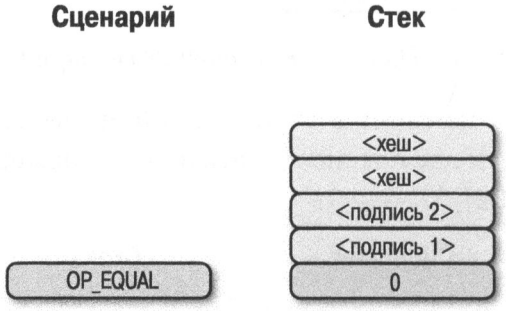

Рис. 8.18. Третья стадия выполнения сценария p2sh

Рис. 8.19. Конечное состояние сценария p2sh, если он вычислялся программным обеспечением, разработанным до появления протокола BIP0016

Если же узлы действуют по протоколу BIP0016, а на момент написания этой книги таких узлов в сети биткойна было подавляющее большинство, то сценарий погашения RedeemScript будет синтаксически проанализирован как последовательность команд Script (рис. 8.20).

```
5221022626e955ea6ea6d98850c994f9107b036b1334f18ca8830bfff129
5d21cfdb702103b287eaf122eea69030a0e9feed096bed8045c8b98bec45
3e1ffac7fbdbd4bb7152ae

- 52 - OP_2
- 21 - Длина <открытого ключа 1>
- 02...db70 - <открытый ключ 1>
- 21 - Длина <открытого ключа 2>
- 03...bb71 - <открытый ключ 1>
- 52 - OP_2
- ae - OP_CHECKMULTISIG
```

Рис. 8.20. Сценарий RedeemScript для p2sh

Эти команды Script указаны в столбце Сценарий на рис. 8.21.

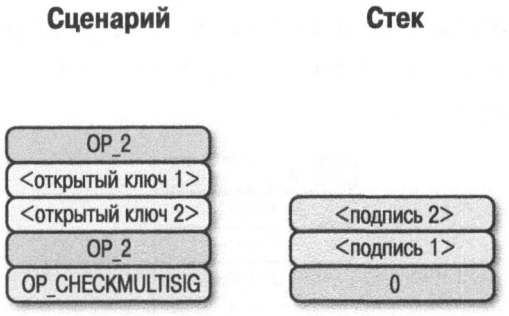

Рис. 8.21. Четвертая стадия выполнения сценария p2sh

Операция OP_2 размещает в стеке число 2 и открытые ключи, а в завершение — еще одно число 2 (рис. 8.22).

Рис. 8.22. Пятая стадия выполнения сценария p2sh

Операция OP_CHECKMULTISIG извлекает из стека $m + n + 3$ элементов, т.е. все содержимое стека. А результат оказывается таким же, как и при выполнении простой мультиподписи (рис. 8.23).

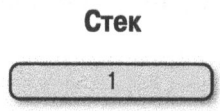

Рис. 8.23. Конечное состояние сценария p2sh, если он вычислялся программным обеспечением, разработанным по протоколу BIP0016

Подстановка сценария RedeemScript выглядит не совсем изящно, поэтому в программном обеспечении биткойна на этот особый случай имеется специальный код. А почему не было выбрано более изящное и понятное решение? На тот момент, когда применялась операция OP_EVAL, протокол BIP0012 был конкурирующим предложением, считавшимся более изящным. И такой сценарий ScriptPubKey, как приведенный на рис. 8.24, должен был нормально выполняться по протоколу BIP0012. Операция OP_EVAL должна была извлечь элемент из вершины стека и интерпретировать его как команды Script, указанные в столбце на рис. 8.24.

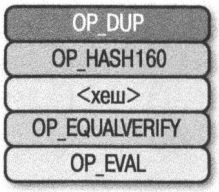

Рис. 8.24. Операция OP_EVAL *должна была служить командой для ввода дополнительных команд, исходя из того, какой именно элемент находился на вершине стека*

Но, к сожалению, более изящному решению сопутствует нежелательный побочный эффект: полнота по Тьюрингу. А полнота по Тьюрингу нежелательна потому, что она существенно затрудняет обеспечение безопасности умного контракта (см. главу 6). Поэтому в протоколе BIP0016 было выбрано менее изящное, но более безопасное решение для особого случая. Протокол BIP0016 (или p2sh) был реализован в 2011 году и продолжает ныне оставаться частью сети биткойна.

Программная реализация p2sh

Специальная последовательность команд в сценарии RedeemScript (OP_HASH160, hash160 и OP_EQUAL) требует обработки. Ее обработку можно организовать в методе evaluate() из исходного файла script.py, как показано ниже.

```
class Script:
...
  def evaluate(self, z):
  ...
  while len(commands) > 0:
    command = commands.pop(0)
    if type(command) == int:
...
    else:
      stack.append(cmd)
      if len(cmds) == 3 and cmds[0] == 0xa9 and type(cmds[1])
                  == bytes and len(cmds[1])
                  == 20 and cmds[2] == 0x87:     ❶
      cmds.pop()     ❷
      h160 = cmds.pop()
      cmds.pop()
      if not op_hash160(stack):     ❸
        return False
        stack.append(h160)
      if not op_equal(stack):
        return False
      if not op_verify(stack):     ❹
        LOGGER.info('bad p2sh h160')
        return False
     redeem_script = encode_varint(len(cmd)) + cmd     ❺
     stream = BytesIO(redeem_script)
     cmds.extend(Script.parse(stream).cmds)     ❻
```

❶ Код 0xa9 обозначает операцию OP_HASH160, а код 0x87 — операцию OP_EQUAL. Здесь проверяется, относятся ли последующие три команды к специальной последовательности команд по протоколу BIP0016.

❷ Известно, что это операция OP_HASH160, поэтому она просто извлекается из стека. Кроме того, известно, что следующей командой является 20-байтовый хеш-код, а третьей командой — операция OP_EQUAL, как и было проверено в приведенном выше условном операторе if.

❸ Здесь выполняется операция OP_HASH160, затем 20-байтовый хеш-код размещается в стеке и, как обычно, выполняется операция OP_EQUAL.

❹ В стеке должна остаться 1, что и проверяется в методе `op_verify()` (операция `OP_VERIFY` извлекает из стека один элемент, но ничего не размещает в нем обратно).

❺ Для синтаксического анализа сценария RedeemScript необходимо предварить его установленной длиной.

❻ Здесь набор команд дополняется синтаксически проанализированными командами из сценария RedeemScript.

Более сложные сценарии

Сценарий p2sh примечателен тем, что сценарий RedeemScript может быть таким же длинным, как и самый крупный элемент из операции `OP_PUSHDATA2` длиной 520 байтов. Мультиподпись является лишь одной возможностью. Ведь возможны сценарии с более сложной логикой вроде "2 из 3 одних ключей или 5 из 7 других ключей". Главной особенностью сценария p2sh является гибкость и в то же время небольшой размер множества UTXO, что накладывает бремя сохранения части сценария на пользователя. Сценарий p2sh будет использован и в главе 13 для обеспечения обратной совместимости протокола Segwit.

Адреса

Чтобы вычислить адреса p2sh, воспользуемся процессом, аналогичным вычислению адресов p2pkh. Для этого хеш-код hash160 предваряется префиксным байтом и дополняется контрольной суммой.

В сети mainnet применяется байт `0x05`, вследствие чего адреса p2sh начинаются с 3 в кодировке Base58, тогда как в сети testnet — байт `0xc4`, из-за чего адреса p2sh начинаются с 2. Адрес можно вычислить с помощью функции `encode_base58_checksum()` из исходного файла `script.py`, как показано ниже.

```
>>> from helper import encode_base58_checksum
>>> h160 = bytes.fromhex('74d691da1574e6b3c192ecfb52cc8984ee7b6c56')
>>> print(encode_base58_checksum(b'\x05' + h160))
3CLoMMyuoDQTPRD3XYZtCvgvkadrAdvdXh
```

Упражнение 2

Напишите функцию `h160_to_p2pkh_address()`, преобразующую 20-байтовый хеш-код hash160 в адрес p2pkh.

Упражнение 3

Напишите функцию `h160_to_p2sh_address()`, преобразующую 20-байтовый хеш-код hash160 в адрес p2sh.

Верификация подписей в p2sh

Как и в p2pkh, одна из трудностей p2sh состоит в верификации подписей. Верификация подписей в p2sh отличается от аналогичного процесса в p2pkh, описанного в главе 7. Но в отличие от p2pkh, где имеются лишь одна подпись и один открытый ключ, в p2sh имеется целый ряд открытых ключей (формата SEC в сценарии RedeemScript) и равное или меньшее количество подписей (формата DER в сценарии ScriptSig). Правда, подписи должны следовать в том же самом порядке, что и открытые ключи, а иначе подписи нельзя считать действительными.

Если у нас есть конкретная подпись и открытый ключ, нам потребуется лишь хеш подписи (z), чтобы проверить эту подпись на достоверность (рис. 8.25).

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e4789e6bd304d87221a00000
0db00483045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e
02205a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a899370148304502210
0da6bee3c93766232079a01639d07fa869598749729ae323eab8eef53577d611b02207bef15429dca
dce2121ea07f233115c6f09034c0be68db99980b9a6c5e754022014752210226 26e955ea6ea6d9885
0c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9feed096bed
8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff04d3b11400000000001976a914904a49878c0
adfc3aa05de7afad2cc15f483a56a88ac7f40090000000001976a914418327e3f3dda4cf5b908932
5a4b95abdfa0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac2224eebdafd1028a
d2788acdc4ace020000000017a91474d691da1574e6b3c192ecfb52cc8984ee7b6c5687 00000000
```

Рис. 8.25. Проверка достоверности вводов p2sh

Как и в p2pkh, выявление хеша подписи оказывается самой трудной частью процесса верификации подписей в p2sh. Поэтому рассмотрим данный процесс более подробно.

Шаг 1. Опорожнение всех полей ScriptSig

Первый шаг состоит в том, чтобы опорожнить все поля ScriptSig, проверяя подпись на достоверность (рис. 8.26). Аналогичная процедура применяется и для создания подписи.

Рис. 8.26. Опорожнение поля ScriptSig каждого ввода

Шаг 2. Замена сценария ScriptSig из подписываемого ввода p2sh сценарием RedeemScript

У каждого ввода p2sh имеется сценарий RedeemScript. Именно этот сценарий и заменяет пустое поле сценария ScriptSig (рис. 8.27). И этим данный шаг отличается от аналогичного шага в p2pkh, где заменяющим является сценарий ScriptPubKey.

Рис. 8.27. Замена сценария ScriptSig из проверяемого ввода сценарием RedeemScript

Шаг 3. Присоединение типа хеша

На последнем шаге в конце присоединяется 4-байтовый тип хеша, как это делается и в p2pkh. В частности, целое число 1, соответствующее типу хеша `SIGHASH_ALL`, должно быть закодировано 4 байтами в прямом порядке их следования, и благодаря этому транзакция выглядит так, как показано на рис. 8.28.

Рис. 8.28. Присоединение типа хеша (`SIGHASH_ALL`), закодированного 4 байтами `01000000`, в самом конце

Чтобы получить хеш подписи *z*, хеш-код hash256 этой модифицированной транзакции следует интерпретировать как целое число, представленное

байтами в обратном порядке их следования. Ниже показано, каким образом хеш-код hash256 модифицированной транзакции преобразуется в хеш подписи *z* на языке Python.

```
>>> from helper import hash256
>>> modified_tx = bytes.fromhex('0100000001868278ed6ddfb6c1ed3ad5f\
8181eb0c7a385aa0836f01d5e4789e6bd304d87221a000000475221022626e955ea\
6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf1\
22eea69030a0e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152affffffff\
04d3b11400000000001976a914904a49878c0adfc3aa05de7afad2cc15f483a56a88\
ac7f40090000000001976a914418327e3f3dda4cf5b9089325a4b95abdfa0334088\
ac722c0c00000000001976a914ba35042cfe9fc66fd35ac2224eebdafd1028ad2788\
acdc4a ce020000000017a91474d691da1574e6b3c192ecfb52cc8984ee7b6c5687\
00000000010000000')
>>> s256 = hash256(modified_tx)
>>> z = int.from_bytes(s256, 'big')
>>> print(hex(z))
0xe71bfa115715d6fd33796948126f40a8cdd39f187e4afb03896795189fe1423c
```

А теперь, когда имеется хеш подписи *z*, можно взять открытый ключ в формате SEC и подпись в формате DER из полей сценариев ScriptSig и RedeemScript (рис. 8.29).

```
0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385aa0836f01d5e4789e6bd304d87221a00000
0db00483045022100dc92655fe37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e
02205a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a899370148304502210
0da6bee3c93766232079a01639d07fa869598749729ae323eab8eef53577d611b02207bef15429dca
dce2121ea07f233115c6f09034c0be68db99980b9a6c5e75402201475221022626e955ea6ea6d9885
0c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9feed096bed
8045c8b98bec453e1ffac7fbdbd4bb7152affffffff04d3b11400000000001976a914904a49878c0
adfc3aa05de7afad2cc15f483a56a88ac7f40090000000001976a914418327e3f3dda4cf5b908932
5a4b95abdfa0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac2224eebdafd1028a
d2788acdc4ace020000000017a91474d691da1574e6b3c192ecfb52cc8984ee7b6c568700000000
```

Рис. 8.29. Подпись формата DER и открытый ключ формата SEC в полях сценариев ScriptSig и RedeemScript для p2sh

Выполним верификацию подписи следующим образом:

```
>>> from ecc import S256Point, Signature
>>> from helper import hash256
>>> modified_tx = bytes.fromhex('0100000001868278ed6ddfb6c1ed3ad5f\
8181eb0c7a385aa0836f01d5e4789e6bd304d87221a000000475221022626e955\
ea6ea6d98850c994f9107b036b1334f18ca8830bfff1295d21cfdb702103b287ea\
f122eea69030a0e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffff\
ffff04d3b11400000000001976a914904a49878c0adfc3aa05de7afad2cc15f483\
a56a88ac7f40090000000001976a914418327e3f3dda4cf5b9089325a4b95abdf\
a0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac2224eebdafd\
1028ad2788acdc4ace020000000017a91474d691da1574e6b3c192ecfb52cc8984\
```

```
ee7b6c56870000000001000000')
>>> h256 = hash256(modified_tx)
>>> z = int.from_bytes(h256, 'big')
>>> sec = bytes.fromhex('022626e955ea6ea6d98850c994f9107b036b1334f18\
ca8830bfff1295d21cfdb70')
>>> der = bytes.fromhex('3045022100dc92655fe37036f47756db8102e0d7d5e\
28b3beb83a8fef4f5dc0559bddfb94e02205a36d4e4e6c7fcd16658c50783e00c3416\
09977aed3ad00937bf4ee942a89937')
>>> point = S256Point.parse(sec)
>>> sig = Signature.parse(der)
>>> print(point.verify(z, sig))
True
```

Таким образом, мы произвели верификацию двух подписей, которые требуются для разблокировки данной мультиподписи в p2sh.

Упражнение 4

Проверьте на достоверность вторую подпись из предыдущей транзакции.

Упражнение 5

Видоизмените методы `sig_hash()` и `verify_input()` таким образом, чтобы производить верификацию транзакций в p2sh.

Заключение

Из этой главы вы узнали, как создавать сценарии ScriptPubKey для p2sh и погашать их. В этой главе были описаны транзакции, упоминавшиеся в последних четырех главах. А в следующей главе поясняется, каким образом они группируются в блоки.

ГЛАВА 9
Блоки

Транзакции передают биткойны от одной стороны к другой, и для этого они разблокируются или разрешаются подписями. Этим гарантируется, что отправитель транзакции разрешил ее. Но что если отправитель отсылает одни и те же монеты по нескольким адресатам? Владелец денежного ящика может попытаться отослать один и тот же вывод дважды. И в этом состоит так называемая *проблема двойного расходования* (double-spending problem). Подобно выдаче банковского чека, от оплаты можно отказаться, получателю транзакции необходимо удостовериться в ее достоверности.

Именно здесь и проявляется действие *блоков* — главного нововведения в биткойне. Блоки можно рассматривать как средство упорядочения транзакций. Так, если упорядочить транзакции, можно тем самым предотвратить двойное расходование, признав любую последующую конфликтующую транзакцию недостоверной. Это все равно что признать предыдущую транзакцию достоверной.

Реализовать такое правило (достоверность первой транзакции и недостоверность последующих транзакций, вступающих с ней конфликт) будет нетрудно, если попытаться упорядочить транзакции по очереди. Но, к сожалению, для этого необходимо, чтобы узлы в сети пришли к общему согласию, какую именно транзакцию следует считать следующей, для чего потребуются немалые затраты на передачу данных. Можно также попытаться упорядочить крупные партии транзакций (возможно, один раз в сутки), но и этот способ вряд ли окажется практичным, поскольку транзакции устраиваются один раз в сутки, а перед тем не имеют завершенности.

В биткойне найдена золотая середина между этими крайними мерами, которая состоит в организации транзакций в партии каждые 10 минут. Именно эти партии транзакций и называются блоками. В этой главе поясняется, каким образом осуществляется синтаксический анализ блоков для проверки

подтверждения работы. И начнем мы со специальной транзакции, которая называется монетизирующей и является первой в каждом блоке.

Монетизирующие транзакции

Монетизирующие (coinbase) транзакции не имеют ничего общего с американской компанией Coinbase. Монетизирующая транзакция должна быть непременно первой в каждом блоке и единственной для производства биткойнов на свет. Выводы монетизирующей транзакции хранятся теми, кто для этого назначен субъектом добычи криптовалюты, и обычно включают в себя всю оплату за другие транзакции в блоке и так называемое *вознаграждение за блок*.

Монетизирующая транзакция служит именно тем стимулом, который побуждает к добыче криптовалюты. На рис. 9.1 показано, каким образом выглядит монетизирующая транзакция.

```
01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff
f5e03d71b07254d696e656420627920416e74506f6f6c20626a31312f4542312f4144362f43205914
293101fabe6d6d678e2c8c34afc36896e7d9402824ed38e856676ee94bfdb0c6c4bcd8b2e5666a040
0000000000000c7270000a5e00e00ffffffff01faf20b58000000001976a914338c84849423992471
bffb1a54a8d9b1d69dc28a88ac00000000

- 01000000 - версия
- 01 - количество входов
- 000...00 - хеш предыдущей транзакции
- ffffffff - индекс предыдущей транзакции
- 5e0...00 - ScriptSig
- ffffffff - последовательность
- 01 - # количество выходов
- faf20b58...00 - сумма на выходе
- 1976...ac - p2pkh ScriptPubKey
- 00000000 - время блокировки
```

Рис. 9.1. Монетизирующая транзакция

Структура монетизирующей транзакции отличается от остальных транзакций в сети биткойна лишь следующими условиями.

1. У монетизирующей транзакции должен быть ровно один ввод.
2. На единственном вводе монетизирующей транзакции должен быть 32-байтовый хеш-код `00` предыдущей транзакции.
3. На единственном вводе монетизирующей транзакции должен быть индекс `ffffffff` предыдущей транзакции.

Именно эти три условия и определяют, является ли транзакция монетизирующей.

Упражнение 1

Напишите метод `is_coinbase()` для класса `Tx`, в котором определяется, является ли транзакция монетизирующей.

Сценарий ScriptSig

У монетизирующей транзакции отсутствует ссылка на расходуемый вывод из предыдущей транзакции, и поэтому на ее вводе ничто не разблокируется. Какой же тогда у нее сценарий ScriptSig?

Сценарий ScriptSig монетизирующей транзакции устанавливается тем, кто добыл эту транзакцию. Главное ограничение состоит в том, что длина сценария ScriptSig должна быть не меньше 2 байтов, но не больше 100 байтов. Помимо этого ограничения и поддержки протокола BIP0034, описываемого в следующем разделе, добытчик криптовалюты может задать сценарий ScriptSig каким угодно при условии, что его вычисление без соответствующего сценария ScriptPubKey подтвердит его достоверность. В качестве примера ниже приведен сценарий ScriptPubKey для монетизирующей транзакции первичного блока.

```
4d04ffff001d0104455468652054696d65732030332f4a616e2f323030392043\
68616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696\
c6f757420666f722062616e6b73
```

Этот сценарий ScriptSig был составлен Сатоши Накамото и содержит сообщение, которое можно воспроизвести следующим образом:

```
>>> from io import BytesIO
>>> from script import Script
>>> stream = BytesIO(bytes.fromhex('4d04ffff001d010445546865205469\
6d65732030332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6\
b206f66207365636f6e64206261696c6f757420666f722062616e6b73'))
>>> s = Script.parse(stream)
>>> print(s.cmds[2])
b'The Times 03/Jan/2009 Chancellor on brink of second bailout for banks'[1]
```

Это сообщение взято из заголовка на первой странице газеты "Таймс" от 3 января 2009 года. Оно подтверждает, что первичный блок был создан именно

[1] Газета "Таймс" от 3.01.2009: "Канцлер на пороге второй государственной помощи на санацию банков".

в *этот* день или некоторое время *спустя*, но не *прежде*. Сценарии других монетизирующих транзакций аналогично содержат произвольные данные.

Протокол BIP0034

Этот протокол устанавливает первый элемент сценария ScriptSig монетизирующей транзакции. Он появился для решения возникшей в сети проблемы, когда добытчики криптовалюты пользовались *одной и той же* монетизирующей транзакцией в разных блоках.

Одинаковый побайтовый характер монетизирующих транзакций означает, что одинаковыми становятся и их идентификаторы, поскольку хеш-код hash256 транзакции детерминирован. Чтобы предотвратить дублирование идентификаторов транзакций, Гэвин Андерсен разработал протокол BIP0034, т.е. правило нерадикального изменения (или так называемой мягкой вилки), вводящее высоту добываемого блока в первый элемент сценария ScriptSig монетизирующей транзакции.

Высота блока интерпретируется как целое число в прямом порядке следования составляющих его байтов. Она должна быть равна количеству блоков, появившихся после первичного блока. Следовательно, монетизирующая транзакция не может быть побайтно одинаковой в разных блоках, поскольку у них разная высота. Ниже показано, каким образом можно осуществить синтаксический анализ высоты блока из монетизирующей транзакции, приведенной на рис. 9.1.

```
>>> from io import BytesIO
>>> from script import Script
>>> from helper import little_endian_to_int
>>> stream = BytesIO(bytes.fromhex('5e03d71b07254d696e656420627920416\
e74506f6f6c20626a31312f4542312f4144362f43205914293101fabe6d6d678e2c8c\
34afc36896e7d9402824ed38e856676ee94bfdb0c6c4bcd8b2e5666a04000000000000\
00c7270000a5e00e00'))
>>> script_sig = Script.parse(stream)
>>> print(little_endian_to_int(script_sig.cmds[0]))
465879
```

Монетизирующая транзакция обнаруживает тот блок, в котором она присутствовала! У монетизирующих транзакций в разных блоках должны быть разные сценарии ScriptSig, а следовательно, и разные идентификаторы. Потребность в этом правиле продолжает оставаться и поныне, ведь иначе идентификаторы монетизирующих транзакций будут дублироваться в разных блоках.

Упражнение 2

Напишите метод `coinbase_height()` для класса `Tx`, в котором определяется высота блока монетизирующей транзакции.

Заголовки блоков

Если блоки являются партиями транзакций, то заголовок блока — метаданными о транзакциях, включаемых в блок. Заголовок блока приведен на рис. 9.2 и состоит из следующих элементов.

- Версия
- Предыдущий блок
- Корень дерева Меркла
- Отметка времени
- Биты данных
- Одноразовый номер

```
020000208ec39428b17323fa0ddec8e887b4a7c53b8c0a0
a220cfd0000000000000000005b0750fce0a889502d4050
8d39576821155e9c9e3f5c3157f961db38fd8b25be1e77a
759e93c0118a4ffd71d

        - 02000020 - версия (4 байта в прямом порядке их следования)
        - 8ec3...00 - предыдущий блок (32 байта в прямом порядке их следования)
        - 5b07...be - корень дерева Меркла (32 байта в прямом порядке их следования)
        - 1e77a759 - отметка времени (4 байта в прямом порядке их следования)
        - e93c0118 - биты данных (4 байта)
        - a4ffd71d - разовый номер (4 байта)
```

Рис. 9.2. Синтаксически проанализированный блок и его заголовок

В заголовке блока содержатся метаданные для блока. В отличие от транзакций, каждое поле в заголовке блока имеет фиксированную длину, как показано на рис. 9.2, а в целом заголовок блока занимает 800 байтов. На момент написания этой книги в сети биткойна имелось около 550000 блоков, а их заголовки занимали около 45 Мбайтов. Весь же блокчейн занимает около 200 Гбайтов, а заголовки блоков — около 0,023% всего этого объема. То обстоятельство, что заголовки намного меньше самих блоков, имеет большое значение, как станет ясно при обсуждении вопросов верификации упрощенных платежей в главе 11.

Подобно идентификатору транзакции, идентификатор блока представлен хеш-кодом hash256 заголовка в прямом порядке следования байтов. Идентификатор блока выглядит весьма любопытно, как показано ниже.

```
>>> from helper import hash256
>>> block_hash = hash256(bytes.fromhex('020000208ec39428b17323fa\
0ddec8e887b4a7c53b8c0a0a220cfd0000000000000000005b0750fce0a88950\
2d40508d39576821155e9c9e3f5c3157f961db38fd8b25be1e77a759e93c0118\
a4ffd71d'))[::-1]
>>> block_id = block_hash.hex()
>>> print(block_id)
00000000000000007e9e4c586439b0cdbe13b1370bdd9435d76a644d047523
```

Именно этот идентификатор указывается при вызове метода `prev_block()` для построения нового блока на основании данного блока, как поясняется далее, а до тех пор лишь заметим, что идентификатор блока начинается со многих нулей. Мы еще вернемся к этому вопросу в разделе "Подтверждение работы", как только рассмотрим поля в заголовке блока более подробно.

Упражнение 3

Напишите метод `parse()` для класса `Block`, чтобы реализовать в нем синтаксический анализ блока.

Упражнение 4

Напишите метод `serialize()` для класса `Block`, чтобы реализовать в нем сериализацию блока.

Упражнение 5

Напишите метод `hash()` для класса `Block`, чтобы реализовать в нем хеширование блока.

Версия

В обычном программном обеспечении *версия* обозначает конкретный ряд функциональных средств. Аналогично содержимое поля версии отражает функциональные возможности программного обеспечения, производящего блок. В прошлом поле версии служило для обозначения одного функционального средства, готового для развертывания добытчиком блока. В частности, версия 2 означала, что программное обеспечение готово поддерживать

протокол BIP0034, в котором было внедрено упоминавшееся ранее в этой главе свойство высоты блока монетизирующей транзакции. Версия 3 означала, что программное обеспечение готово поддерживать протокол BIP0066, в котором соблюдалось строгое кодирование в формате DER. А версия 4 означала, что программное обеспечение готово поддерживать протокол BIP0065, в котором была определена операция `OP_CHECKLOCKTIMEVERIFY`.

Но, к сожалению, постепенное увеличение номера версии означало также, что одновременно в сети могло быть задействовано лишь одно функциональное средство. Чтобы ослабить данное ограничение, разработчики придумали протокол BIP0009, позволяющий одновременно задействовать в сети до 29 различных функциональных средств.

Протокол BIP0009 действует таким образом, чтобы зафиксировать первые 3 бита 4-байтового (32-разрядного) заголовка блока равными `001` и тем самым указать на то, что добытчик криптовалюты применяет протокол BIP0009. Благодаря тому, что первые 3 бита заголовка равны `001`, прежним клиентам приходится интерпретировать содержимое поля версии как номер версии, больший или равный 4 (т.е. номеру последней версии, использовавшейся до появления протокола BIP0009).

Это означает, что в шестнадцатеричной форме первым символом всегда будет 2 или 3, а остальные 29 битов могут быть присвоены другим функциональным средствам мягкой вилки, о готовности задействовать которые могут оповестить добытчики криптовалюты. Например, бит 0 (т.е. крайний справа младший бит) может быть установлен в 1, чтобы оповестить о готовности к одной мягкой вилке, бит 1 (т.е. второй справа бит) может быть установлен в 1, чтобы оповестить о готовности к другой мягкой вилке. А бит 2 (т.е. третий справа бит) может быть установлен в 1, чтобы оповестить о готовности к следующей мягкой вилке и т.д.

Протокол BIP0009 требует, чтобы 95% всех блоков оповещали о готовности в заданной эпохе из 2016 блоков (т.е. периода корректировки сложности; подробнее об этом — далее в главе). На момент написания этой книги в протокол BIP0009 были внесены следующие мягкие вилки (т.е. нерадикальные изменения): BIP0068/BIP0112/BIP0113 (операция `OP_CHECKSEQUENCEVERIFY` и связанные с ней изменения) и BIP0141 (Segwit). В этих протоколах BIP биты 0 и 1 использовались для оповещения о соответствующих мягких вилках. Протокол BIP0091 применяется аналогично протоколу BIP0009, но с 80%-ным порогом и меньшим периодом для блоков, а следовательно, он не был

в строгом смысле таким же, как и протокол BIP0009. Для оповещения протокола BIP0091 служил бит 4.

Проверить все упомянутые выше средства нетрудно, как показано ниже.

```
>>> from io import BytesIO
>>> from block import Block
>>> b = Block.parse(BytesIO(bytes.fromhex('020000208ec39428b1\
7323fa0ddec8e887b4a7c53b8c0a0a220cfd0000000000000000005b0750\
fce0a889502d40508d39576821155e9c9e3f5c3157f961db38fd8b25be1e\
77a759e93c0118a4ffd71d')))
>>> print('BIP9: {}'.format(b.version >> 29 == 0b001))     ❶
BIP9: True
>>> print('BIP91: {}'.format(b.version >> 4 & 1 == 1))     ❷
BIP91: False
>>> print('BIP141: {}'.format(b.version >> 1 & 1 == 1))     ❸
BIP141: True
```

❶ Операция >> служит для поразрядного сдвига вправо, в результате которого отбрасываются крайние справа 29 бит, а в итоге остаются лишь 3 старших бита. Здесь `0b001` обозначает номер версии в двоичной форме, принятой в Python.

❷ Операция `&` служит для поразрядного логического "И". В данном случае сначала сдвигаются вправо первые 4 бита, а затем проверяется, установлена ли 1 в крайнем справа младшем бите.

❸ Здесь 1 сдвигается вправо, поскольку для бита 1 назначен протокол BIP0141.

Упражнение 6

Напишите метод `bip9()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP0009.

Упражнение 7

Напишите метод `bip91()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP00091.

Упражнение 8

Напишите метод `bip141()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP0141.

Предыдущий блок

Все блоки должны указывать на предыдущий блок. Именно поэтому рассматриваемая здесь структура данных называется *блокчейном* (т.е. цепочкой блоков). Все блоки связаны по цепочке, ведущей обратно к самому первому блоку, иначе называемому *первичным*. Поле предыдущего блока завершается последовательностью нулевых (00) байтов, о чем пойдет речь далее в этой главе.

Корень дерева Меркла

В корне дерева Меркла все упорядоченные транзакции кодируются в виде 32-байтового хеш-кода. О том, насколько это важно для клиентов с упрощенной проверкой оплаты (SPV) и как они пользуются корнем дерева Меркла с данными из сервера для получения подтверждения включения данных транзакции в корень этого дерева, речь пойдет в главе 11.

Отметка времени

Отметка времени делается в формате Unix и занимает до 4 байтов. Такая отметка времени обозначает количество секунд, истекших с 1 января 1970 года, т.е. с начала так называемой *эпохи* в Unix. Она применяется в двух целях: для проверки достоверности блокировок времени в транзакциях, включаемых в блок, а также для вычисления битов, цели (т.е. заданной величины) и сложности через каждые 2016 блоков. Одно время блокировки времени применялись в блоке непосредственно для транзакций, но этот режим был изменен в протоколе BIP0113, чтобы вместо отметки времени непосредственно из блока пользоваться средним прошедшим временем (MTP) 11 последних блоков.

Произойдет ли переполнение биткойна по отметке времени

Поле отметки времени в заголовке блока биткойна занимает 4 байта или 32 бита. Это означает, что однажды отметка времени в формате Unix превысит величину $2^{32} - 1$ и для ее хранения больше не останется места. 2^{32} секунд приблизительно равно 136 годам, а это означает, что в данном поле больше не останется места к 2106 году, т.е. через 136 лет после 1970 года.

Многие ошибочно считают, что это произойдет в 2038 году, т.е. через 68 лет после 1970 года, но такое может случиться, лишь если поле отметки времени относится к целочисленному типу со знаком. И хотя 2^{31} секунд составляет 68 лет, в данном случае предельный срок наступит в 2106 году благодаря использованию дополнительного бита знака. А в 2106 году для заголовка блока потребуется особого рода вилка, поскольку ему больше некуда будет постоянно расти.

Биты

В поле битов кодируется подтверждение работы, требующееся в данном блоке. Подробнее об этом речь пойдет в следующем разделе.

Одноразовый номер

В этом поле содержится используемый лишь один раз номер (*nonce*). Он изменяется добытчиками криптовалюты в поисках подтверждения работы.

Подтверждение работы

Именно подтверждение работы защищает биткойн, а на более глубоком уровне оно позволяет производить децентрализованную добычу биткойнов. Поиск подтверждения работы дает добытчику криптовалюты право присоединить блок к цепочке блоков в блокчейне. С одной стороны, подтверждение работы происходит очень редко, и поэтому перед добытчиком криптовалюты стоит непростая задача. А с другой стороны, подтверждение работы проверяется объективно и просто, и поэтому добытчиком криптовалюты может стать всякий желающий.

Подтверждение работы называется "добычей" по весьма веской причине. Как и при настоящей добыче ископаемых, именно этого ищут добытчики криптовалюты. При добыче золота из 45 тонн породы обычно получается 1 унция (0,02835 кг) этого драгоценного металла. Именно поэтому золото так ценится. Но как только золото добыто, проверить его подлинность совсем нетрудно. Для этого имеются химические анализы, пробирные камни и многие другие средства и способы, позволяющие относительно недорого определить подлинность золота.

Аналогично подтверждение работы — это число, обеспечивающее весьма редкий результат. Чтобы найти подтверждение работы, добытчикам в сети биткойна приходится перерабатывать числовой эквивалент породы. Как и при добыче золота, проверить подтверждение работы намного дешевле, чем фактически найти его.

Так что же такое подтверждение работы? Чтобы ответить на этот вопрос, рассмотрим упоминавшийся ранее хеш-код hash256 заголовка блока:

```
020000208ec39428b17323fa0ddec8e887b4a7c53b8c0a0a220cfd\
0000000000000000005b0750fce0a889502d40508d39576821155e9c\
9e3f5c3157f961db38fd8b25be1e77a759e93c0118a4ffd71d
>>> from helper import hash256
>>> block_id = hash256(bytes.fromhex('020000208ec39428b17323\
fa0ddec8e887b4a7c53b8c0a0a220cfd0000000000000000005b0750fce0a\
889502d40508d39576821155e9c9e3f5c3157f961db38fd8b25be1e77a759\
e93c0118a4ffd71d'))[::-1]
>>> print('{}'.format(block_id.hex()).zfill(64))     ❶
0000000000000000007e9e4c586439b0cdbe13b1370bdd9435d76a644d047523
```

❶ Здесь намеренно выводится данное число в виде 64 шестнадцатеричных цифр чтобы показать, насколько оно мало в 256-разрядной форме.

Как известно, хеш-функция hash256 служит для генерирования равномерно распределяемых значений. Принимая это обстоятельство во внимание, два цикла выполнения хеш-функции sha256 или hash256 можно интерпретировать как случайное число. Вероятность, что любое случайное 256-разрядное число окажется очень малым, весьма ничтожна. В частности, вероятность, что первый бит 256-разрядного числа окажется равным нулю, составляет 0,5, два первых его бита окажутся равными нулю (00) — 0,25, три первых бита окажутся равными нулю (000) — 0,125 и т.д. А поскольку каждый 0 в шестнадцатеричной форме представляет четыре нулевых бита, вероятность, что первые 73 бита 256-разрядного числа окажутся равным нулю, составит $0{,}5^{73}$ или 1 на 10^{22} случаев, т.е. она действительно ничтожна. В среднем, чтобы получить искомое число со столь ничтожной вероятностью, придется сгенерировать 10^{22} (или 10 триллионов триллионов) случайных 256-разрядных чисел. Иными словами, для этого придется вычислить в среднем 10^{22} хеш-кодов. Если вернуться к аналогии с добычей золота, то в процессе поиска подтверждения работы придется переработать 10^{22} битов породы, чтобы добыть числовой эквивалент золотого слитка.

Каким образом добытчик криптовалюты генерирует хеш-коды

Откуда добытчик криптовалюты получает новый числовой аналог породы для переработки, чтобы выяснить, удовлетворяет ли он подтверждению работы? Для этой цели служит поле одноразового номера. Добытчики криптовалюты могут самопроизвольно менять содержимое данного поля, чтобы изменить хеш-код заголовка блока.

Но, к сожалению, 4 байтовой или 32-разрядной длины поля одноразового номера (т.е. 2^{32} одноразовых номеров, которые может опробовать добытчик криптовалюты) недостаточно для подтверждения работы. Современное вычислительное оборудование на основе специализированных интегральных микросхем (ASIC) способно вычислить намного больше, чем 2^{32} разных хеш-кодов в секунду. Например, блок добычи AntMiner S9 способен вычислить 12 терахеш-кодов в секунду, т.е. около 2^{43} хеш-кодов в секунду. А это означает, что все пространство одноразовых номеров может быть использовано всего лишь за 0,0003 секунды.

Что же делать добытчикам криптовалюты в том случае, если поле одноразового номера исчерпается? В таком случае они могут изменить монетизирующую транзакцию, а следовательно, и корень дерева Меркла, чтобы получить совершенно новое пространство одноразовых номеров. А с другой стороны, они могут свернуть поле версии или воспользоваться открытым алгоритмом оптимизации ASICBOOST. Внутренний механизм изменений в корне дерева Меркла при изменении любой транзакции в блоке рассматривается в главе 11.

Цель

Подтверждение работы является требованием, что хеш-код заголовка каждого блока в биткойне должен быть меньше определенной цели. *Цель* — это заданное 256-разрядное число, вычисляемое непосредственно из поля битов (в данном примере — `e93c0118`). Цель весьма мала в сравнении со средним 256-разрядным числом.

В поле битов фактически содержатся два разных числа. Первое из них — показатель степени, хранящийся в последнем байте, а второе — коэффициент, хранящийся в трех других байтах, следующих в прямом порядке. Ниже приведена формула для вычисления цели из этих двух целей.

$$\text{цель} = \text{коэффициент} \times 256^{\text{показатель степени} - 3}$$

Если задано поле битов, то цель вычисляется в Python следующим образом:

```
>>> from helper import little_endian_to_int
>>> bits = bytes.fromhex('e93c0118')
>>> exponent = bits[-1]
>>> coefficient = little_endian_to_int(bits[:-1])
>>> target = coefficient * 256**(exponent - 3)
>>> print('{:x}'.format(target).zfill(64))    u
0000000000000000013ce9000000000000000000000000000000000000000000
```

❶ Здесь намеренно выводится данное число в виде 64 шестнадцатеричных цифр с целью показать, насколько оно мало в 256-разрядной форме.

Достоверное подтверждение работы представляет собой хеш-код заголовка блока, который после интерпретации в виде целого числа с прямым порядком следования байтов оказывается меньше заданного числа (т.е. ниже заданной цели). Хеш-коды подтверждения работы крайне редки, а процесс добычи криптовалюты, собственно, состоит в поиске одного из этих хеш-кодов. Чтобы найти одно подтверждение работы по предыдущей цели, в целой сети биткойна придется вычислить 3.8×10^{21} хеш-кодов, что после обнаружения данного блока может быть сделано приблизительно через каждые 10 минут. Чтобы поставить такое число в какой-то реальный контекст, следует заметить, что самому лучшему в мире графическому процессору для добычи криптовалюты потребуется в среднем 50000 лет, чтобы обнаружить подтверждение работы нижезаданной цели.

Проверить, удовлетворяет ли хеш-код заголовка блока искомому подтверждению работы, можно следующим образом:

```
>>> from helper import little_endian_to_int
>>> proof = little_endian_to_int(hash256(bytes.fromhex(
'020000208ec39428b17323fa0ddec8e887b4a7c53b8c0a0a220cfd\
0000000000000000005b0750fce0a889502d40508d39576821155\
e9c9e3f5c3157f961db38fd8b25be1e77a759e93c0118a4ffd71d')))
>>> print(proof < target)     ❶
True
```

❶ Заданная цель (`target`) вычисляется в приведенном выше фрагменте кода.

Если сопоставить рядом полученные выше числа (цели и идентификатора блока соответственно) в виде 64 шестнадцатеричных символов, то можно заметить, что подтверждение работы оказывается ниже заданной цели:

```
TG: 0000000000000000013ce9000000000000000000000000000000000000000000

ID: 0000000000000000007e9e4c586439b0cdbe13b1370bdd9435d76a644d047523
```

Упражнение 9

Напишите функцию `bits_to_target()` из исходного файла `helper.py`, чтобы вычислить цель из заданных битов.

Сложность

Понять назначение цели простым смертным непросто. Цель — это всего лишь число, меньше которого должен быть вычисленный хеш-код, но людям непросто заметить, чем 180-разрядное число отличается от 190-разрядного. И хотя первое из них в тысячу раз меньше второго, глядя на столь большие числа, очень трудно поставить их в нужный контекст.

Для того чтобы упростить сравнение разных целей, было придумано понятие *сложности*. Смысл в том, чтобы сделать сложность обратно пропорциональной цели и тем самым упростить сравнение. А ниже приведена формула, по которой определяется их соотношение.

$$\text{сложность} = 0\text{xffff} \times 256^{0\text{x1d}-3} / \text{цель}$$

Непосредственно в коде сравнение целей осуществляется следующим образом:

```
>>> from helper import little_endian_to_int
>>> bits = bytes.fromhex('e93c0118')
>>> exponent = bits[-1]
>>> coefficient = little_endian_to_int(bits[:-1])
>>> target = coefficient*256**(exponent-3)
>>> difficulty = 0xffff * 256**(0x1d-3) / target
>>> print(difficulty)
888171856257.3206
```

Сложность биткойна в первичном блоке составляла 1. Этим предоставляется необходимый контекст для оценки сложности сети mainnet в настоящий момент. Сложность можно рассматривать как средство, позволяющее выяснить, насколько добыть криптовалюту теперь труднее, чем изначально. В частности, добыть криптовалюту в приведенном выше коде приблизительно в 888 миллиардов раз труднее, чем на начальной стадии существования биткойна. Сложность нередко отображается в обозревателях биткойна и службах построения графиков котировки биткойна, чтобы упростить оценку тех усилий, которые потребуются для создания нового блока.

Упражнение 10

Напишите метод `difficulty()` для класса `Block`, чтобы определить в нем сложность добычи.

Проверка достаточности подтверждения работы

Итак, мы уже выяснили, что подтверждение работы может быть вычислено в результате расчета хеш-кода hash256 заголовка блока и его интерпретации в виде целого числа с прямым порядком следования байтов. Если полученное в итоге число оказывается меньше заданной цели, то в нашем распоряжении имеется достоверное подтверждение работы. В противном случае блок нельзя считать достоверным, поскольку у него нет подтверждения работы.

Упражнение 11

Напишите метод `check_pow()` для класса `Block`, чтобы проверить в нем подтверждение работы.

Корректировка сложности

В биткойне каждая группа из 2016 блоков называется *периодом корректировки сложности*. В конце каждого периода корректировки сложности цель корректируется по следующим формулам.

> *разность_времени* = (отметка времени последнего блока в период корректировки сложности) – (отметка времени первого блока в период корректировки сложности)
>
> *новая_цель* = *предыдущая_цель* * *разность_времени* / (2 недели)

Здесь *разность_времени* вычисляется следующим образом: если она больше 8 недель, то становится равной 8 неделям, а если она меньше 3,5 дней, то приравнивается к 3,5 дням. Таким образом, новая цель не может измениться больше, чем в четыре раза, в ту или иную сторону. Это означает, что цель будет понижена или повышена не больше чем в четыре раза.

Если для создания каждого блока в среднем требуется 10 минут, то для создания 2016 блоков — 20160 минут. И если учесть, что в сутках 1440 минут, то для создания 2016 блоков потребуется 20160 / 1440 = 14 дней. А в результате корректировки сложности время на создание каждого блока в среднем сокращается до 10 минут. Это означает, что в долгосрочной перспективе

время создания блоков будет всегда сводиться к 10 минутам, даже если в сеть вводятся или выводятся из нее немалые вычислительные мощности для хеширования.

Новые биты должны вычисляться с помощью поля отметки времени из последнего блока в каждый текущий или предыдущий период корректировки сложности. Но, к сожалению, в рассматриваемой здесь формуле Сатоши Накамото проявляется еще одна ошибка смещения на единицу, поскольку при вычислении разности отметок времени рассматриваются первый и последний блоки из периода корректировки сложности 2016 блоков. Следовательно, разность времени определяется как разность отметок времени 2015, а не 2016 блоков.

Рассматриваемую здесь формулу можно запрограммировать следующим образом:

```
>>> from block import Block
>>> from helper import TWO_WEEKS     ❶
>>> last_block = Block.parse(BytesIO(bytes.fromhex(
'00000020fdf740b0e49cf75bb3d5168fb3586f7613dcc5cd89675b\
0100000000000000002e37b144c0baced07eb7e7b64da916cd\
3121f2427005551aeb0ec6a6402ac7d7f0e4235954d801187f5da9f5')))
>>> first_block = Block.parse(BytesIO(bytes.fromhex(
'000000201ecd89664fd205a37566e694269ed76e425803003628ab\
0100000000000000bfcade29d080d9aae8fd461254b0418\
05ae442749f2a40100440fc0e3d5868e55019345954d80118a1721b2e')))
>>> time_differential = last_block.timestamp - first_block.timestamp
>>> if time_differential > TWO_WEEKS * 4:     ❷
... time_differential = TWO_WEEKS * 4
>>> if time_differential < TWO_WEEKS // 4:     ❸
... time_differential = TWO_WEEKS // 4
>>> new_target = last_block.target() * time_differential // TWO_WEEKS
>>> print('{:x}'.format(new_target).zfill(64))
0000000000000000007615000000000000000000000000000000000000000000
```

❶ Следует заметить, что в выражении `TWO_WEEKS = 60*60*24*14` определяется количество секунд в 2 неделях: 60 секунд × 60 минут × 24 часа × 14 дней.

❷ Здесь проверяется следующее: если для обнаружения 2015 последних блоков потребовалось больше 8 недель, то сложность повышается ненамного.

❸ А здесь проверяется следующее: если для обнаружения 2015 последних блоков потребовалось меньше 3,5 дней, то и тогда сложность повышается ненамного.

Следует заметить, что для вычисления цели следующего блока требуются только заголовки. И как только будет вычислена искомая цель, ее можно будет преобразовать обратно в биты. Эта обратная операция реализуется в коде следующим образом:

```
def target_to_bits(target):
'''Преобразует целочисленное значение цели обратно в биты'''
raw_bytes = target.to_bytes(32, 'big')
raw_bytes = raw_bytes.lstrip(b'\x00')     ❶
if raw_bytes[0] > 0x7f:     ❷
exponent = len(raw_bytes) + 1
coefficient = b'\x00' + raw_bytes[:2]
else:
exponent = len(raw_bytes)     ❸
coefficient = raw_bytes[:3]     ❹
new_bits = coefficient[::-1] + bytes([exponent])     ❺
return new_bits
```

❶ Избавиться от всех начальных нулей.

❷ Формат битов служит для выражения очень больших чисел (как положительных, так и отрицательных) в краткой форме. Так, если первый бит в коэффициенте равен 1, содержимое поля битов интерпретируется как отрицательное число. А поскольку цель всегда положительна, все содержимое поля битов сдвигается на 1 байт, если первый его бит равен 1.

❸ Показатель степени, определяющей длину числа по основанию 256.

❹ Коэффициент, составляющий первые три цифры числа по основанию 256.

❺ Коэффициент, представленный байтами в прямом порядке их следования, а также показатель степени в формате битов.

Если в блоке отсутствуют подходящие биты, вычисленные по формуле корректировки сложности, такой блок можно благополучно отвергнуть.

Упражнение 12

Вычислите новые биты при условии, что первый и последний из 2016 блоков в период корректировки сложности таковы.

- Блок 471744

```
000000203471101bbda3fe307664b3283a9ef0e97d9a38a7eacd88\
0000000000000000010c8aba8479bbaa5e0848152fd3c2289ca50\
e1c3e58c9a4faaafbdf5803c5448ddb845597e8b0118e43a81d3
```

- Блок 473759

```
02000020f1472d9db4b563c35f97c428ac903f23b7fc055d1cfc26\
0000000000000000b3f449fcbe1bc4cfbcb8283a0d2c037f961\
a3fdf2b8bedc144973735eea707e1264258597e8b0118e5f00474
```

Упражнение 13

Напишите функцию `calculate_new_bits()` из исходного файла `helper.py` для вычисления новых битов.

Заключение

В этой главе рассматривалось, каким образом рассчитывается подтверждение работы, вычисляются новые биты для блока по истечении периода корректировки сложности и осуществляется синтаксический анализ монетизирующих транзакций. А теперь можно перейти к вопросам организации сети в главе 10, продвигаясь по пути к еще не описанному полю корня дерева Меркла из заголовка блока, что и будет сделано в главе 11.

ГЛАВА 10

Организация сети

Надежность биткойна обеспечивается одноранговой сетью, в которой он действует. На момент написания данной книги в этой сети действовало больше 65 тысяч постоянно взаимодействующих узлов.

Сеть биткойна является широковещательной или циркулярной. В каждом ее узле объявляются разные транзакции, блоки и те одноранговые узлы, которые ему известны. В этой сети действует полноценный протокол, который с годами был дополнен многими функциональными возможностями.

Следует, однако, иметь в виду, что сетевой протокол некритичен к согласованию. Одни и те же данные могут быть отправлены из одного узла в другой по совсем иному протоколу, не оказывая никакого влияния на сам блокчейн. Принимая все это во внимание, мы рассмотрим в этой главе вопросы запроса, получения и проверки достоверности заголовков блоков по сетевому протоколу.

Сетевые сообщения

Все сетевые сообщения выглядят так, как на рис. 10.1. Первые 4 байта всегда одинаковы и называются *сетевыми волшебными байтами*. Волшебные байты типичны для сетевого программирования, поскольку обмен данными осуществляется в асинхронном режиме и может прерываться. Волшебные байты предоставляют принимающей сообщение стороне место, с которого можно продолжить обмен данными, прерванный, например, из-за пропадания сигнала в телефонной линии. Они полезны и для выявления сети, например для того, чтобы не соединять узел биткойна с узлом лайткойна, поскольку у последнего совсем другие волшебные байты. В сети testnet также используются другие волшебные байты (`0b110907`), в отличие от сети mainnet, в которой они равны `f9beb4d9`.

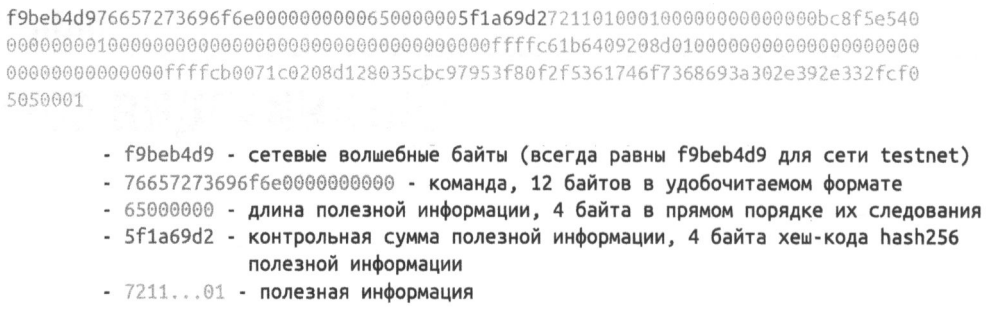

Рис. 10.1. Сетевое сообщение, в конверт которого вкладывается полезная информация

Следующие 12 байтов составляют поле команды или описание того, что фактически переносится по сети. Имеются и многие другие команды, исчерпывающий перечень которых приведен в документации к сетевому протоколу (`https://en.bitcoin.it/wiki/Protocol_documentation`). Данные в поле команды представлены в удобочитаемом виде, а в рассматриваемом здесь сообщении оно содержит строку, дополненную нулевыми байтами в коде ASCII.

Следующие четыре байта обозначают длину полезной информации в прямом порядке их следования. Как пояснялось в главах 5 и 9, длину полезной информации требуется указывать потому, что сама полезная информация переменна. Максимальная длина составляет 2^{32} или около 4 миллиардов, а следовательно, объем полезной информации может достигать 4 Гбайтов, хотя эталонный клиент отвергает любые полезные данные свыше 32 Мбайтов. В сообщении, приведенном на рис. 10.1, длина полезной информации составляет 101 байтов.

Последующие 4 байта обозначают поле контрольной суммы. Выбор алгоритма вычисления контрольной суммы кажется несколько странным, поскольку его результат размещается в первых 4 байтах хеш-кода hash256 полезной информации. Странность такого выбора объясняется тем, что контрольные суммы сетевого протокола, как правило, служат для исправления ошибок, тогда как в алгоритме hash256 этого не требуется. Но поскольку алгоритм hash256 применяется в остальной части биткойна, именно по этой причине он применяется и здесь.

Чтобы реализовать в исходном коде обработку сетевых сообщений, потребуется новый класс, как показано ниже.

```
NETWORK_MAGIC = b'\xf9\xbe\xb4\xd9'
TESTNET_NETWORK_MAGIC = b'\x0b\x11\x09\x07'

class NetworkEnvelope:

  def __init__(self, command, payload, testnet=False):
    self.command = command
    self.payload = payload
    if testnet:
      self.magic = TESTNET_NETWORK_MAGIC
    else:
      self.magic = NETWORK_MAGIC

  def __repr__(self):
    return '{}: {}'.format(
      self.command.decode('ascii'),
      self.payload.hex(),
    )
```

Упражнение 1

Напишите метод `parse()` для класса `NetworkEnvelope`, чтобы реализовать синтаксический анализ сетевого сообщения.

Упражнение 2

Выясните, что именно содержит следующее сетевое сообщение:

```
f9beb4d976657261636b000000000000000000005df6e0e2
```

Упражнение 3

Напишите метод `serialize()` для класса `NetworkEnvelope`, чтобы реализовать сериализацию содержимого сетевого сообщения.

Синтаксический анализ полезной информации

Для каждой команды имеется отдельная спецификация полезной информации. На рис. 10.2 приведен результат синтаксического анализа полезной информации о команде `version`.

Поля в сообщении, приведенном на рис. 10.2, служат для предоставления достаточных сведений, чтобы организовать обмен данными между двумя узлами. В первом поле данного сообщения содержится версия сетевого

протокола, обозначающая, какими именно сообщениями можно обмениваться между узлами сети. В следующем далее поле сетевой службы предоставляются сведения о возможностях, доступных для соединения узлов сети. Длина поля отметки времени составляет 8 байтов, в отличие от 4 байтов в заголовке блока, и в этом поле хранится отметка времени в формате Unix с прямым порядком следования байтов.

```
7f11010000000000000000ad17835b0000000000000000000000000000000000ffff0
0000008d2000000000000000000000000000000000ffff00000008d20f6a8d7a440ec27a11b
2f70726f6772616d6d696e67626c6f636b636861696e3a302e312f0000000001

7f110100 - Версия протокола, 4 байта в прямом порядке их следования
0000000000000000 - Сетевые службы отправителя, 8 байтов в прямом порядке их следования
ad17835b00000000 - Отметка времени, 8 байтов в прямом порядке их следования
0000000000000000 - Сетевые службы получателя, 8 байтов в прямом порядке их следования
00000000000000000000ffff00000000 - Сетевой адрес получателя, 16 байтов
        0.0.0.0
8d20 - B208d — Сетевой порт получателя, 2 байта
0000000000000000 - Сетевые службы отправителя, 8 байтов в прямом порядке их следования
00000000000000000000ffff00000000 - Сетевой адрес отправителя, 16 байтов
        0.0.0.0
8d20 - B208d — Сетевой порт отправителя, 2 байта
f6a8d7a440ec27a1 - Разовый номер, 8 байтов, служит для передачи ответов
1b2f70726f6772616d6d696e67626c6f636b636861696e3a302e312f - Пользовательский посредник
        /программированиебиткойна: 0.1/
00000000 - Высота, 0
01 - Дополнительный признак для пересылки, основанный на протоколе BIP37
```

Рис. 10.2. Результат синтаксического анализа полезной информации о команде `version`

IP-адреса могут быть заданы в формате протокола IPv6, IPv4 или адаптера виртуальной частной сети OnionCat, преобразующего адреса из псевдодомена `.onion` сети TOR в формат протокола IPv6. Если эти адреса заданы в формате протокола IPv4, то первые 12 байтов равны `00000000000000000000ffff`, а последние 4 байта составляют IP-адрес. Для номера сетевого порта отведено 2 байта в прямом порядке их следования. Стандартно в сети mainnet задается номер порта `8333`, преобразующийся в шестнадцатеричное значение `208d` с прямым порядком следования байтов.

В поле одноразового номера указано число, используемое узлом для обнаружения соединения с ним, а пользовательский посредник выявляет выполняемое программное обеспечение. Поле высоты или последнего блока помогает другому узлу выяснить, с каким блоком он синхронизирован. И наконец, поле пересылки служит для задания фильтров Блума, о которых речь пойдет в главе 12.

Ниже приведен класс `VersionMessage`, в котором задаются стандартные значения в полях рассмотренного выше сообщения.

```
class VersionMessage:
  command = b'version'

  def __init__(self, version=70015, services=0, timestamp=None,
               receiver_services=0,
               receiver_ip=b'\x00\x00\x00\x00', receiver_port=8333,
               sender_services=0,
               sender_ip=b'\x00\x00\x00\x00', sender_port=8333,
               nonce=None, user_agent=b'/programmingbitcoin:0.1/',
               latest_block=0, relay=False):
    self.version = version
    self.services = services
    if timestamp is None:
      self.timestamp = int(time.time())
    else:
      self.timestamp = timestamp
    self.receiver_services = receiver_services
    self.receiver_ip = receiver_ip
    self.receiver_port = receiver_port
    self.sender_services = sender_services
    self.sender_ip = sender_ip
    self.sender_port = sender_port
    if nonce is None:
      self.nonce = int_to_little_endian(randint(0, 2**64), 8)
    else:
      self.nonce = nonce
    self.user_agent = user_agent
    self.latest_block = latest_block
    self.relay = relay
```

А теперь необходимо выяснить, каким образом сериализировать данное сообщение.

Упражнение 4

Напишите метод `serialize()` для класса `VersionMessage`, чтобы реализовать в нем сериализацию сообщения о версии сетевого протокола.

Подтверждение подключения к сети

Подтверждение подключения к сети означает порядок установления связи между узлами, как поясняется ниже.

- Допустим, что узлу A требуется подключиться к узлу B, и поэтому он отсылает сообщение о версии сетевого протокола.
- Узел B принимает сообщение о версии сетевого протокола, отвечает сообщением, подтверждающим версию сетевого протокола, и отсылает свое сообщение о версии сетевого протокола.
- Узел A принимает сообщения о версии сетевого протокола и ее подтверждении и отсылает обратно свое сообщение, подтверждающее версию сетевого протокола.
- Узел B принимает сообщение, подтверждающее версию сетевого протокола, и продолжает обмен данными с узлом A.

Как только описанная выше процедура подтверждения подключения к сети завершится, узлы A и B смогут обмениваться данными как им угодно. Следует, однако, иметь в виду, что здесь отсутствует аутентификация, и поэтому ответственность за верификацию всех получаемых данных возлагается на узлы сети. Если же узел отошлет неудачную транзакцию или блок, то он может быть запрещен или отключен.

Подключение к сети

Трудности передачи данных по сети объясняются ее асинхронным характером. В качестве эксперимента можно попробовать установить соединение с узлом сети синхронно, как показано ниже.

```
>>> import socket
>>> from network import NetworkEnvelope, VersionMessage
>>> host = 'testnet.programmingbitcoin.com'     ❶
>>> port = 18333
>>> socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> socket.connect((host, port))
>>> stream = socket.makefile('rb', None)     ❷
>>> version = VersionMessage()     ❸
>>> envelope = NetworkEnvelope(version.command, version.serialize())
>>> socket.sendall(envelope.serialize())     ❹
>>> while True:
...     new_message = NetworkEnvelope.parse(stream)     ❺
...     print(new_message)
```

❶ Это сервер, установленный для сети testnet. По умолчанию в сети testnet устанавливается порт 18333.

❷ Здесь создается поток, в который можно читать данные из сетевого сокета. Созданный таким образом поток может быть передан всем методам синтаксического анализа.

❸ Первая стадия подтверждения подключения к сети состоит в том, чтобы отправить сообщение о версии сетевого протокола.

❹ А теперь сообщение отсылается в правильном конверте.

❺ В этой строке кода будут прочитаны любые сообщения, поступающие через подключенный сетевой сокет.

Если подключиться к сети подобным образом, то отправить свое сообщение не удастся до тех пор, пока не будет получено сообщение от другого узла, а также разумно отвечать всего лишь на одно сообщение одновременно. Для более надежной реализации отправки и приема сообщений по сети без блокировки можно воспользоваться асинхронной библиотекой (вроде `asyncio` в версии Python 3).

Необходимо также создать класс для реализации сообщения, подтверждающего версию сетевого протокола, как показано ниже. В классе `VerAckMessage` реализуется минимальное сетевое сообщение.

```
class VerAckMessage:
  command = b'verack'

  def __init__(self):
    pass
    @classmethod

  def parse(cls, s):
    return cls()

  def serialize(self):
    return b''
```

А теперь автоматизируем данный процесс, создав класс, в котором реализуется обмен данными по сети.

```
class SimpleNode:

  def __init__(self, host, port=None, testnet=False, logging=False):
    if port is None:
      if testnet:
        port = 18333
      else:
        port = 8333
```

```
    self.testnet = testnet
    self.logging = logging
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.socket.connect((host, port))
    self.stream = self.socket.makefile('rb', None)

  def send(self, message):    ❶
    '''Отправить сообщение подключенному узлу'''
    envelope = NetworkEnvelope(
      message.command, message.serialize(), testnet=self.testnet)
    if self.logging:
      print('sending: {}'.format(envelope))
    self.socket.sendall(envelope.serialize())

  def read(self):    ❷
    '''Прочитать сообщение из сетевого сокета'''
    envelope = NetworkEnvelope.parse(
              self.stream, testnet=self.testnet)
    if self.logging:
      print('receiving: {}'.format(envelope))
    return envelope

  def wait_for(self, *message_classes):    ❸
    '''Ожидать одно из сообщений, заданных в списке'''
    command = None
    command_to_class = {m.command: m for m in message_classes}
    while command not in command_to_class.keys():
      envelope = self.read()
      command = envelope.command
      if command == VersionMessage.command:
        self.send(VerAckMessage())
      elif command == PingMessage.command:
        self.send(PongMessage(envelope.payload))
      return command_to_class[command].parse(envelope.stream())
```

❶ Метод `send()` отправляет сообщение через сетевой сокет. При этом предполагается, что свойство `command` и метод `serialize()` существуют в объекте `message`.

❷ Метод `read()` читает новое сообщение из сетевого сокета.

❸ Метод `wait_for()` позволяет ожидать любую из нескольких команд (в частности, из классов, реализующих сообщения). Помимо синхронного характера данного класса, подобный метод упрощает программирование в целом. В узлах коммерческого уровня ничего подобного, определенно, не применяется.

А теперь, когда у нас имеется узел в сети, можно реализовать подтверждение подключения к другому узлу, как показано ниже.

```
>>> from network import SimpleNode, VersionMessage
>>> node = SimpleNode('testnet.programmingbitcoin.com', testnet=True)
>>> version = VersionMessage()     ❶
>>> node.send(version)     ❷
>>> verack_received = False
>>> version_received = False
>>> while not verack_received and not version_received:     ❸
...   message = node.wait_for(VersionMessage, VerAckMessage)     ❹
...   if message.command == VerAckMessage.command:
...     verack_received = True
...   else:
...     version_received = True
...     node.send(VerAckMessage())
```

❶ Большинство узлов не интересуют поля вроде IP-адреса из объекта, присваиваемого переменной `version`. Подключиться можно и со стандартно установленными полями.

❷ Подтверждение подключения к сети начинается с отправки сообщения о версии сетевого протокола.

❸ Подтверждение подключения к сети завершится лишь в том случае, если будут получены сообщения о версии сетевого протокола и подтверждении подключения.

❹ Здесь предполагается получить от другого узла сообщение, подтверждающее подключение к сети, а также сообщение об его версии сетевого протокола, хотя заранее неизвестно, в каком именно порядке они поступят.

Упражнение 5

Напишите метод `handshake()` для класса `SimpleNode`, чтобы реализовать в нем подтверждение подключения к сети.

Получение заголовков блоков

Что еще можно сделать теперь, когда у нас имеется код для подключения к узлу сети? Когда любой узел подключается к сети в первый раз, наиболее критичными для получения и верификации данными становятся заголовки блоков. Полные узлы получают возможность загрузить заголовки блоков, чтобы асинхронно запросить полные блоки из многих узлов, распараллелив

загрузку блоков. А "тонкие" клиенты могут загрузить заголовки, чтобы проверить подтверждение работы в каждом блоке. Как поясняется в главе 11, "тонкие" клиенты будут в состоянии получать подтверждение включения через сеть, но для этого им потребуются заголовки блоков.

Узлы могут предоставить заголовки блоков, не занимая много пропускной способности сети. Для получения заголовков блоков служит команда getheaders, структура которой приведена на рис. 10.3.

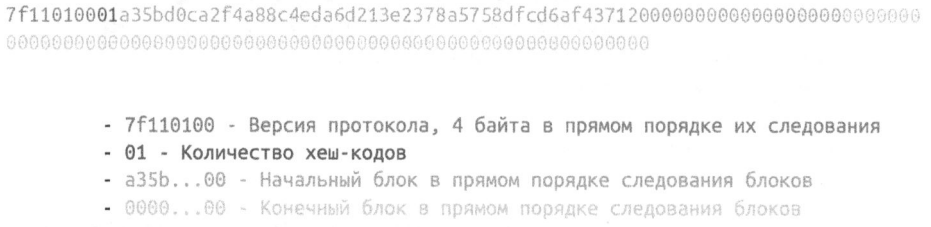

Рис. 10.3. Результат синтаксического анализа команды для получения заголовков блоков

Как и при согласовании версии сетевого протокола, данное сообщение начинается с версии сетевого протокола, затем следует количество групп заголовков блоков в списке (это количество может быть больше 1, если цепочка разделена), далее — заголовок начального блока и наконец — заголовок конечного блока. Если же указать заголовок конечного блока равным 000...000, это означает, что требуется получить столько заголовков блоков, сколько сможет предоставить другой узел. Максимальное количество заголовков, которое можно получить обратно, равно 2000, т.е. приближается к периоду корректировки сложности (2016 блоков).

Ниже приведен класс, реализующий получение заголовков блоков.

```
class GetHeadersMessage:
  command = b'getheaders'

  def __init__(self, version=70015, num_hashes=1,
    start_block=None, end_block=None):
    self.version = version
    self.num_hashes = num_hashes     ❶
    if start_block is None:     ❷
      raise RuntimeError('a start block is required')
    self.start_block = start_block
    if end_block is None:
      self.end_block = b'\x00' * 32     ❸
    else:
      self.end_block = end_block
```

❶ Для простоты здесь предполагается, что количество групп заголовков блоков равно 1. А в более надежной реализации придется обрабатывать не одну такую группу, хотя заголовки блоков можно загрузить, используя единственную группу.

❷ Здесь требуется начальный блок, иначе создать надлежащее сообщение не удастся.

❸ Здесь предполагается, что конечный блок пустой. Если же он не определен, то предполагается получить столько заголовков блоков, сколько отправит сервер.

Упражнение 6

Напишите метод `serialize()` для класса `GetHeadersMessage`, чтобы реализовать в нем сериализацию сообщения о получении заголовков блоков.

Присылаемые в ответ заголовки

А теперь можно создать узел, подтвердить его подключение к сети и запросить ряд заголовков блоков, как показано ниже.

```
>>> from io import BytesIO
>>> from block import Block, GENESIS_BLOCK
>>> from network import SimpleNode, GetHeadersMessage
>>> node = SimpleNode('mainnet.programmingbitcoin.com', testnet=False)
>>> node.handshake()
>>> genesis = Block.parse(BytesIO(GENESIS_BLOCK))
>>> getheaders = GetHeadersMessage(start_block=genesis.hash())
>>> node.send(getheaders)
```

Теперь требуется каким-то образом получить заголовки из другого узла, который отправит обратно команду `headers` со списком заголовков блоков (рис. 10.4), синтаксический анализ которого рассматривался в главе 9. Этим синтаксическим анализом можно выгодно воспользоваться в классе `HeadersMessage`.

```
0200000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef
2ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e6700
0000002030eb2540c41025690160a1014c577061596e32e426b712c7ca00000000000000768b89f07
044e6130ead292a3f51951adbd2202df447d98789339937fd006bd44880835b67d8001ade0920460 0
```

- 02 - Количество заголовков блоков
- 00...67 - Заголовок блока
- 00 - Количество транзакций (всегда равно 0)

Рис. 10.4. Результат синтаксического анализа списка заголовков блоков

Сообщение о заголовках блоков начинается с количества заголовков типа `varint` в пределах от 1 до 2000 включительно. Как известно, длина каждого заголовка блока составляет 80 байтов. Затем следует количество транзакций, которое в данном сообщении всегда равно нулю. Поначалу это может вызвать недоумение, поскольку запрашивались только заголовки, но не транзакции. Узлы сети заинтересованы в отправке количества транзакций потому, что рассматриваемое здесь сообщение о заголовках блоков должно быть совместимо по формату с сообщением о блоках, в котором указываются заголовок блока, количество транзакций и сами транзакции. Указав нулевое количество транзакций в рассматриваемом здесь сообщении, можно воспользоваться тем же самым механизмом синтаксического анализа, что и для полного блока, как показано ниже.

```
class HeadersMessage:
  command = b'headers'

  def __init__(self, blocks):
    self.blocks = blocks

  @classmethod
  def parse(cls, stream):
    num_headers = read_varint(stream)
    blocks = []
    for _ in range(num_headers):
      blocks.append(Block.parse(stream))     ❶
      num_txs = read_varint(stream)     ❷
      if num_txs != 0:     ❸
        raise RuntimeError('number of txs not 0')
    return cls(blocks)
```

❶ Каждый блок синтаксически анализируется в методе `parse()` из класса `Block` с помощью того же самого потока, который уже имеется.

❷ Количество транзакций всегда равно нулю и является остатком от синтаксического анализа блоков.

❸ Если не получен нуль, значит, что-то не так.

Как только сетевое соединение будет установлено, появится возможность загрузить заголовки блоков, проверить в них подтверждение работы и достоверность корректировок сложности, как показано ниже.

```
>>> from io import BytesIO
>>> from network import SimpleNode, GetHeadersMessage, HeadersMessage
>>> from block import Block, GENESIS_BLOCK, LOWEST_BITS
```

```
>>> from helper import calculate_new_bits
>>> previous = Block.parse(BytesIO(GENESIS_BLOCK))
>>> first_epoch_timestamp = previous.timestamp
>>> expected_bits = LOWEST_BITS
>>> count = 1
>>> node = SimpleNode('mainnet.programmingbitcoin.com', testnet=False)
>>> node.handshake()
>>> for _ in range(19):
...     getheaders = GetHeadersMessage(start_block=previous.hash())
...     node.send(getheaders)
...     headers = node.wait_for(HeadersMessage)
...     for header in headers.blocks:
...       if not header.check_pow():     ❶
...         raise RuntimeError('bad PoW at block {}'.format(count))
...       if header.prev_block != previous.hash():     ❷
...         raise RuntimeError('discontinuous block at {}'
...                            .format(count))
...       if count % 2016 == 0:
...         time_diff = previous.timestamp - first_epoch_timestamp
...         expected_bits = calculate_new_bits(
...                         previous.bits, time_diff)     ❹
...         print(expected_bits.hex())
...         first_epoch_timestamp = header.timestamp     ❺
...       if header.bits != expected_bits:     ❸
...         raise RuntimeError('bad bits at block {}'
...                            .format(count))
...       previous = header
...       count += 1
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
ffff001d
6ad8001d
28c4001d
71be001d
```

❶ Проверить достоверность подтверждение работы.

❷ Проверить, следует ли текущий блок после предыдущего.

❸ Проверить, соответствуют ли биты, цель и сложность ожидаемым величинам, исходя из вычислений в предыдущей эпохе.

❹ Вычислить в конце эпохи следующие биты, цель и сложность.

❺ Сохранить первый блок из эпохи для вычисления битов в конце данной эпохи.

Следует, однако, иметь в виду, что приведенный выше код не будет нормально действовать в сети testnet, поскольку в ней применяется другой алгоритм корректировки сложности. Если блок не был обнаружен в сети testnet в течение 20 минут, корректировка сложности упадет до 1, чтобы обеспечить согласованное и упрощенное обнаружение блоков для тестирования. Именно таким образом тестирующие смогут и дальше создавать блоки в сети testnet без дорогостоящего оборудования для добычи криптовалюты. Так, недорогое вычислительное оборудование USB ASIC стоимостью 30 долларов США способно обнаруживать несколько блоков в минуту при минимальной степени сложности.

Заключение

В этой главе было показано, каким образом осуществляется и подтверждается подключение к узлу в сети, загружаются заголовки блоков и проверяется их соответствие согласованным правилам. А в следующей главе основное внимание будет уделено получению сведений о представляющих интерес транзакциях из узла сети частным, хотя и вполне доказуемым образом.

ГЛАВА 11

Упрощенная проверка оплаты

В главе 9 не было подробно описано лишь поле заголовка, содержащее корень дерева Меркла. Чтобы стали понятнее преимущества корня дерева Меркла, необходимо рассмотреть сначала сами деревья Меркла и их свойства, что и будет сделано в этой главе, ведь к этому побуждает понятие *подтверждение включения*.

Предпосылки

Для устройства, на жестком диске которого имеется немного места, пропускной способности или вычислительной мощности, хранение, получение и проверка достоверности всего блокчейна в целом обходятся недешево. На момент написания данной книги вся цепочка блоков биткойна занимала около 200 Гбайтов, но такой объем данных неспособен храниться на большинстве мобильных телефонов. Чтобы загрузить такое количество информации эффективно, определенно, потребуются немалые вычислительные ресурсы ЦП. Но если весь блокчейн нельзя разместить в мобильном телефоне, что же можно еще сделать? Можно ли создать на мобильном телефоне биткойновый кошелек, не имея в своем распоряжении все эти данные?

Любой кошелек выполняет две основные функции.

1. Выплата другим лицам.
2. Получение платы от других лиц.

Если вы выплачиваете кому-нибудь из своего биткойнового кошелька, то на получателя ваших биткойнов возлагается обязанность убедиться, что ему действительно заплатили. Убедившись, что транзакция была включена в блок достаточно глубоко, он предоставит вам другую сторону торговой сделки, товар или услугу. Как только вы отправите транзакцию другой стороне, вам

не останется ничего другого, как только ждать до тех пор, пока вы получите именно то, что вам требуется, в обмен на свои биткойны.

Но, получая плату биткойнами, мы сталкиваемся с дилеммой. Так, если мы подключены к сети биткойна и имеем в своем распоряжении весь блокчейн, то можем легко заметить, что транзакция находится достаточно глубоко в блоке. И в таком случае мы предоставляем другой стороне свои товары или услуги. Но если в нашем распоряжении отсутствует весь блокчейн, как это имеет место в мобильном телефоне, то что же тогда нам остается делать?

Ответ на этот вопрос кроется в поле корня дерева Меркла из заголовка блока, рассмотренного в главе 9. Как пояснялось в предыдущей главе, заголовки блоков можно загрузить и проверить на соответствие согласованным правилами биткойна. А в этой главе мы продвинемся дальше в сторону получения специального подтверждения, что конкретная транзакция находится в известном нам блоке. Поскольку заголовок блока защищается подтверждением работы, транзакция с подтверждением ее включения в этот блок как минимум означает, что на производство данного блока затрачено значительное количество энергии. Это также означает, что если вас кто-нибудь обманет в блокчейне, такой обман будет стоить ему по меньшей мере подтверждения работы, выполненной для создания блока. В остальной части этой главы поясняется понятие подтверждения включения и способов его проверки.

Дерево Меркла

Дерево Меркла (Merkle tree) — это структура вычислительной техники, предназначенная для эффективного подтверждения включения. В качестве предпосылок для построения дерева Меркла служат упорядоченный список элементов и криптографическая хеш-функция. В данном случае элементами упорядоченного списка являются транзакции, находящиеся в блоке, а также хеш-функция hash256. Чтобы построить дерево Меркла, достаточно выполнить следующие действия.

1. Хешировать все элементы упорядоченного списка с помощью предоставляемой хеш-функции.
2. Завершить операцию, если имеется ровно 1 хеш-код или просто хеш.
3. В противном случае продублировать последний хеш в списке, если количество хешей нечетное, и добавить его в конце списка, чтобы это количество стало четным.

4. Составить хеши парами по порядку и хешировать их, чтобы перейти на родительский уровень, где количество хешей сократится вдвое.
5. Перейти к п. 2.

Смысл в том, чтобы дойти до единственного хеша, "представляющего" весь упорядоченный список. На рис. 11.1 наглядно показано, как выглядит дерево Меркла.

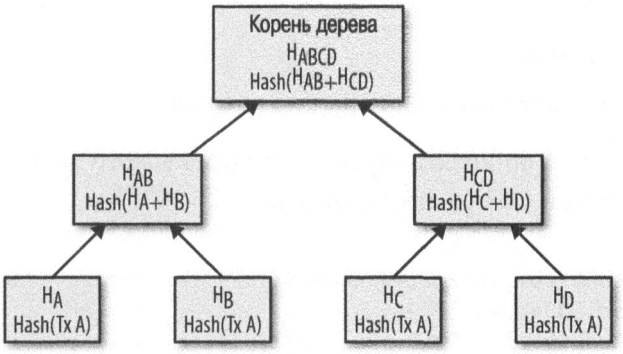

Рис. 11.1. Дерево Меркла

В нижнем ряду блок-схемы, приведенной на рис. 11.1, находятся так называемые *листья* дерева, а все остальные ее части являются *внутренними узлами* дерева Меркла. Листья объединяются на *родительском уровне* (H_{AB} и H_{CD}), на котором образуется корень дерева Меркла. Все эти части дерева Меркла будут подробно рассмотрены в последующих разделах.

Будьте осторожны, манипулируя деревьями Меркла!

В версиях биткойна 0.4–0.6 существовала уязвимость, связанная с корнем дерева Меркла, подробно описанная в документе CVE-2012-2459. Это была уязвимость к атакам типа отказа в обслуживании из-за дублирования последнего элемента в деревьях Меркла, что приводило к тому, что в некоторых узлах блоки признавались недействительными, даже если они на самом деле были действительными.

Родительский узел дерева Меркла

Имея в своем распоряжении два хеша, можно получить третий хеш, который их представляет. А упорядочив оба хеша, один из них можно назвать

левым хешем, а другой — *правым* хешем. Тот же хеш, который получается из левого и правого хешей, называется *родительским*. Ниже для ясности приведена формула, по которой определяется родительский хеш.

- $P=H(L||R)$
- H — функция хеширования
- P — родительский хеш
- L — левый хеш
- R — правый хеш
- $||$ — знак, обозначающий операцию сцепления, или соединения.

Описанный выше процесс получения родительского хеша можно реализовать на языке Python следующим образом:

```
>>> from helper import hash256
>>> hash0 = bytes.fromhex('c117ea8ec828342f4dfb0ad6bd140e03a5072\
0ece40169ee38bdc15d9eb64cf5')
>>> hash1 = bytes.fromhex('c131474164b412e3406696da1ee20ab0fc9bf\
41c8f05fa8ceea7a08d672d7cc5')
>>> parent = hash256(hash0 + hash1)
>>> print(parent.hex())
8b30c5ba100f6f2e5ad1e2a742e5020491240f8eb514fe97c713c31718ad7ecd
```

Результат сцепления двух исходных хешей подвергается хешированию для того, чтобы обеспечить подтверждение включения. В частности, можно показать, что левый хеш L представлен в родительском хеше P, выявив правый хеш R. Так, если нам требуется подтвердить, что левый хеш L представлен в родительском хеше P, поставщик родительского хеша P может показать правый хеш R и дать нам знать, что левый хеш L является производным от родительского хеша P. И тогда мы сможем соединить левый хеш L с правым хешем R, чтобы получить родительский хеш P и тем самым подтвердить, что левый хеш L был действительно использован для получения родительского хеша P. Если же левый хеш L не представлен в родительском хеше P, то возможность получить родительский хеш P будет равнозначна предоставлению прообраза хеша, что, как известно, сделать очень трудно. Вот, собственно, что означает подтверждение включения.

Упражнение 1

Напишите функцию `merkle_parent()`, реализующую получение родительского хеша в дереве Меркла.

Родительский уровень дерева Меркла

Имея в своем распоряжении упорядоченный список более чем из двух хешей, можно вычислить родителей каждой пары хешей или так называемый *родительский уровень дерева Меркла*. Сделать это нетрудно, если имеется четное количество хешей, поскольку их достаточно составить в пары по порядку. Если же имеется нечетное количество хешей, придется сделать нечто большее, поскольку в конечном счете остается один непарный хеш. Это затруднение можно разрешить, продублировав последний элемент в списке хешей.

Это означает, что, имея в своем распоряжении список [*A*, *B*, *C*], можно ввести в него еще один элемент *C*, получив в итоге список [*A*, *B*, *C*, *C*]. И тогда можно вычислить узлы дерева Меркла, являющиеся родительскими для узлов *A* и *B*, а также для узлов *C* и *C*, по следующей формуле:

$$[H(A||B), H(C||C)]$$

А поскольку родительский узел дерева Меркла всегда состоит из двух хешей, на родительском уровне дерева Меркла всегда находится ровно половина дочерних хешей после округления в *большую* сторону. Количество хешей на родительском уровне дерева Меркла вычисляется в коде следующим образом:

```
>>> from helper import merkle_parent
>>> hex_hashes = [
...   'c117ea8ec828342f4dfb0ad6bd140e03a50720ece40169ee38bdc15d9eb64cf5',
...   'c131474164b412e3406696da1ee20ab0fc9bf41c8f05fa8ceea7a08d672d7cc5',
...   'f391da6ecfeed1814efae39e7fcb3838ae0b02c02ae7d0a5848a66947c0727b0',
...   '3d238a92a94532b946c90e19c49351c763696cff3db400485b813aecb8a13181',
...   '10092f2633be5f3ce349bf9ddbde36caa3dd10dfa0ec8106bce23acbff637dae',
... ]
>>> hashes = [bytes.fromhex(x) for x in hex_hashes]
>>> if len(hashes) % 2 == 1:
...   hashes.append(hashes[-1])     ❶
>>> parent_level = []
>>> for i in range(0, len(hashes), 2):     ❷
...   parent = merkle_parent(hashes[i], hashes[i+1])
...   parent_level.append(parent)
>>> for item in parent_level:
...   print(item.hex())
8b30c5ba100f6f2e5ad1e2a742e5020491240f8eb514fe97c713c31718ad7ecd
7f4e6f9e224e20fda0ae4c44114237f97cd35aca38d83081c9bfd41feb907800
3ecf6115380c77e8aae56660f5634982ee897351ba906a6837d15ebc3a225df0
```

❶ Здесь вводится последний хеш в список `hashes` (т.е. `hashes[-1]`), чтобы сделать его длину четной.

❷ Именно таким образом данный цикл выполняется с шагом 2 в Python. На первом шаге значение переменной цикла `i` равно 0, на втором — 2, на третьем — 4 и т.д.

В результате выполнения приведенного выше кода получается новый список, соответствующий родительскому уровню дерева Меркла.

Упражнение 2

Напишите функцию `merkle_parent_level()`, вычисляющую родительский уровень дерева Меркла.

Корень дерева Меркла

Чтобы получить корень дерева Меркла, можно последовательно вычислять родительские уровни дерева Меркла до тех пор, пока не будет получен единственный хеш. Так, если имеются элементы *A*–*G* (т.е. 7 элементов), вычислить первый родительский уровень дерева Меркла можно следующим образом:

$$[H(A||B), H(C||D), H(E||F), H(G||G)]$$

Затем родительский уровень дерева Меркла вычисляется снова:

$$[H(H(A||B)||H(C||D)), H(H(E||F)||H(G||G))]$$

В итоге получаются лишь два элемента, и поэтому родительский уровень дерева Меркла вычисляется еще раз:

$$H(H(H(A||B)||H(C||D))||H(H(E||F)||H(G||G)))$$

На этом процесс вычисления корня дерева Меркла завершается, поскольку в конечном счете остался ровно один хеш. На каждом уровне дерева количество хешей сокращается вдвое, и поэтому неоднократное выполнение данного процесса приведет в конечном счете к единственному узлу, называемому *корнем дерева Меркла*, как показано ниже.

```
>>> from helper import merkle_parent_level
>>> hex_hashes = [
...   'c117ea8ec828342f4dfb0ad6bd140e03a50720ece40169ee38bdc15d9eb64cf5',
...   'c131474164b412e3406696da1ee20ab0fc9bf41c8f05fa8ceea7a08d672d7cc5',
...   'f391da6ecfeed1814efae39e7fcb3838ae0b02c02ae7d0a5848a66947c0727b0',
...   '3d238a92a94532b946c90e19c49351c763696cff3db400485b813aecb8a13181',
...   '10092f2633be5f3ce349bf9ddbde36caa3dd10dfa0ec8106bce23acbff637dae',
```

```
...     '7d37b3d54fa6a64869084bfd2e831309118b9e833610e6228adacdbd1b4ba161',
...     '8118a77e542892fe15ae3fc771a4abfd2f5d5d5997544c3487ac36b5c85170fc',
...     'dff6879848c2c9b62fe652720b8df5272093acfaa45a43cdb3696fe2466a3877',
...     'b825c0745f46ac58f7d3759e6dc535a1fec7820377f24d4c2c6ad2cc55c0cb59',
...     '95513952a04bd8992721e9b7e2937f1c04ba31e0469fbe615a78197f68f52b7c',
...     '2e6d722e5e4dbdf2447ddecc9f7dabb8e299bae921c99ad5b0184cd9eb8e5908',
...     'b13a750047bc0bdceb2473e5fe488c2596d7a7124b4e716fdd29b046ef99bbf0',
... ]
>>> hashes = [bytes.fromhex(x) for x in hex_hashes]
>>> current_hashes = hashes
>>> while len(current_hashes) > 1:     ❶
... current_hashes = merkle_parent_level(current_hashes)
>>> print(current_hashes[0].hex())     ❷
acbcab8bcc1af95d8d563b77d24c3d19b18f1486383d75a5085c4e86c86beed6
```

❶ Здесь выполняется цикл до тех пор, пока не останется один хеш.

❷ А здесь происходит выход из цикла, если останется лишь один хеш.

Упражнение 3

Напишите функцию `merkle_root()`, вычисляющую корень дерева Меркла.

Корень дерева Меркла в блоках

На первый взгляд, вычисление корня дерева Меркла в блоках должно быть выполнено довольно просто, но вследствие обратного порядка следования байтов оно затруднено. В частности, для листьев дерева Меркла применяется прямой порядок следования байтов, а после вычисления корня дерева Меркла этот порядок используется снова.

На практике это означает, что порядок следования байтов в листьях дерева Меркла должен быть изменен на обратный как перед началом вычисления, так и после него. Ниже показано, как это реализовать непосредственно в коде.

```
>>> from helper import merkle_root
>>> tx_hex_hashes = [
...     '42f6f52f17620653dcc909e58bb352e0bd4bd1381e2955d19c00959a22122b2e',
...     '94c3af34b9667bf787e1c6a0a009201589755d01d02fe2877cc69b929d2418d4',
...     '959428d7c48113cb9149d0566bde3d46e98cf028053c522b8fa8f735241aa953',
...     'a9f27b99d5d108dede755710d4a1ffa2c74af70b4ca71726fa57d68454e609a2',
...     '62af110031e29de1efcad103b3ad4bec7bdcf6cb9c9f4afdd586981795516577',
...     '766900590ece194667e9da2984018057512887110bf54fe0aa800157aec796ba',
...     'e8270fb475763bc8d855cfe45ed98060988c1bdcad2ffc8364f783c98999a208',
... ]
```

```
>>> tx_hashes = [bytes.fromhex(x) for x in tx_hex_hashes]
>>> hashes = [h[::-1] for h in tx_hashes]     ❶
>>> print(merkle_root(hashes)[::-1].hex())     ❷
654d6181e18e4ac4368383fdc5eead11bf138f9b7ac1e15334e4411b3c4797d9
```

❶ Порядок следования байтов в каждом хеше обращается перед обработкой списка хешей в Python.

❷ Порядок следования байтов снова обращается после вычисления корня дерева Меркла.

Чтобы вычислить корни деревьев Меркла в классе `Block`, необходимо добавить параметр `tx_hashes`, как показано ниже.

```
class Block:

  def __init__(self, version, prev_block, merkle_root,
               timestamp, bits, nonce, tx_hashes=None):     ❶
    self.version = version
    self.prev_block = prev_block
    self.merkle_root = merkle_root
    self.timestamp = timestamp
    self.bits = bits
    self.nonce = nonce
    self.tx_hashes = tx_hashes
```

❶ Здесь допускается устанавливать хеши транзакций как часть инициализации блока. При этом хеши транзакций должны быть упорядочены.

Имея в своем распоряжении полный узел, если получены все транзакции, можно вычислить корень дерева Меркла и проверить его на соответствие тому, что фактически ожидается.

Упражнение 4

Напишите метод `validate_merkle_root()` для класса `Block`, чтобы реализовать в нем проверку корня дерева Меркла на достоверность.

Применение дерева Меркла

Теперь, когда известно, как построить дерево Меркла, можно создать и проверить на достоверность подтверждения включения. "Легкие" узлы могут получить подтверждения, что интересующие их транзакции были включены в блок, даже не зная обо всех транзакциях в блоке (рис. 11.2).

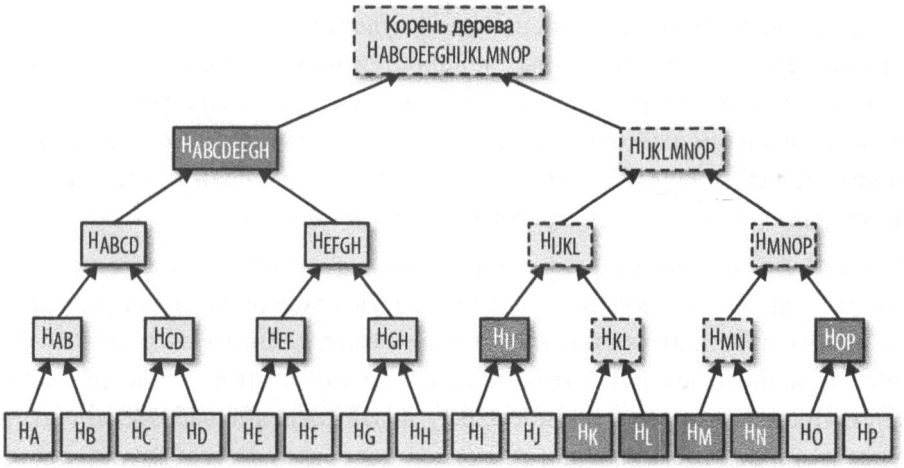

Рис. 11.2. Подтверждение включения в дереве Меркла

Допустим, что у "тонкого" клиента имеются две интересующие его транзакции, т.е. хеши, обозначенные как H_K и H_N на рис. 11.2. В полном узле можно построить подтверждение включения, получив все хеши, обозначенные как $H_{ABCDEFGH}$, H_{IJ}, H_L, H_M и H_{OP}. И тогда в "тонком" клиенте могут быть выполнены следующие вычисления.

- $H_{KL} = merkle\_parent(H_K, H_L)$
- $H_{MN} = merkle\_parent(H_M, H_N)$
- $H_{IJKL} = merkle\_parent(H_{IJ}, H_{KL})$
- $H_{MNOP} = merkle\_parent(H_{MN}, H_{OP})$
- $H_{IJKLMNOP} = merkle\_parent(H_{IJKL}, H_{MNOP})$
- $H_{ABCDEFGHIJKLMNOP} = merkle\_parent(H_{ABCDEFGH}, H_{IJKLMNOP})$

На рис. 11.2 пунктиром обозначены те хеши, которые вычисляются "тонким" клиентом. В частности, корень дерева Меркла, обозначенный как $H_{ABCDEFGHIJKLMNOP}$, может быть проверен на наличие заголовка блока, в котором проверено на достоверность подтверждение работы.

Насколько безопасна упрощенная проверка оплаты

Полный узел может отправить ограниченное количество информации о блоке, а "тонкий" клиент — пересчитать корень дерева Меркла, который может быть затем проверен сравнением с аналогичным корнем в заголовке блока. И хотя этим не гарантируется, что транзакция находится в самой длинной

цепочке блоков, "тонкий" клиент все же удостоверяется, что полный узел затратил немало вычислительных мощностей на хеширование и энергии на формирование подтверждения работы. А поскольку вознаграждение за такое подтверждение работы оказывается больше, чем суммы в транзакциях, "тонкий" клиент будет по крайней мере знать, что у полного узла нет вообще никакой материальной заинтересованности обманывать его.

Заголовки блоков могут быть запрошены из многих узлов, и поэтому у "тонких" клиентов появляется возможность проверить, предпринимается ли в одном узле попытка показать им не самые длинные заголовки блоков. Чтобы признать недействительными сотни нечестных узлов, достаточно единственного честного узла, поскольку подтверждение работы объективно. Следовательно, чтобы совершить подлог подобным способом, потребуется изоляция "тонкого" клиента (т.е. контроль над тем, к каким именно узлам сети он подключен). И поэтому для обеспечения безопасности упрощенной проверки оплаты требуется наличие многих честных узлов в сети.

Иными словами, безопасность "тонкого" клиента основывается на надежности функционирования сети узлов и на экономических затратах на получение подтверждения работы. Если для совершения мелких транзакций относительно блочной ссуды (в настоящее время размером 12,5 биткойнов) это не так важно, то для крупных транзакций (например, на сумму 100 биткойнов) у одной стороны может возникнуть материальная заинтересованность обмануть другую сторону. Поэтому достоверность крупных транзакций должна, как правило, проверяться с помощью полного узла.

Древовидный блок Меркла

Когда полный узел отправляет подтверждение включения, то в него должны быть включены два фрагмента информации. Во-первых, "тонкому" клиенту требуется структура дерева Меркла, а во-вторых, ему необходимо знать, в каких именно узлах дерева Меркла находятся отдельные хеши. И как только оба фрагмента информации будут предоставлены, "тонкий" клиент сможет восстановить дерево Меркла вплоть до его корня и проверить подтверждение включения на достоверность. Полный узел передает оба эти фрагмента информации "тонкому" клиенту, используя древовидный блок Меркла.

Чтобы стало понятнее назначение древовидного блока Меркла, необходимо хотя бы немного пояснить, каким образом осуществляется обход дерева Меркла в частности и двоичных деревьев вообще. В двоичном дереве узлы

обходятся в ширину или в глубину. Обход в ширину обычно происходит от одного уровня к другому, как показано на рис. 11.3. Обход дерева в ширину начинается с его корня и продолжается по его листьям от одного уровня к другому и слева направо.

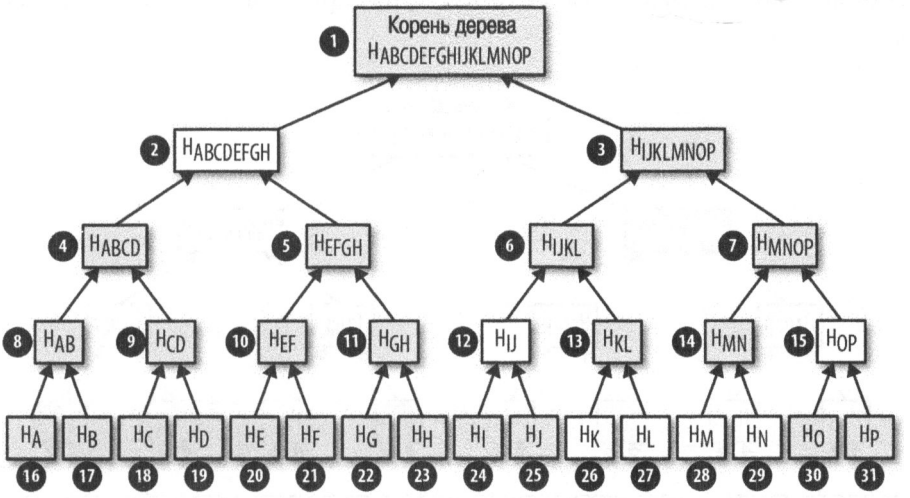

Рис. 11.3. Обход дерева Меркла в ширину

А обход дерева в глубину осуществляется несколько иначе, как показано на рис. 11.4. Обход дерева в глубину начинается с его корня и продолжается от левой стороны каждого узла к правой его стороне.

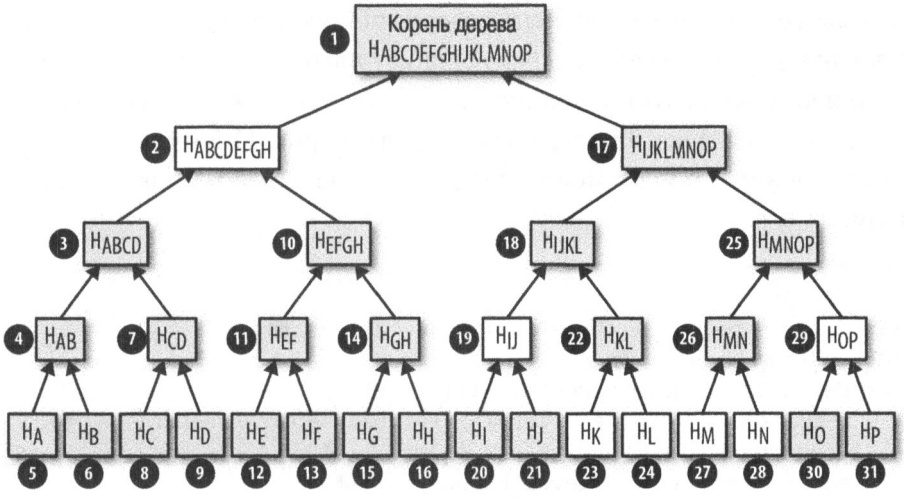

Рис. 11.4. Обход дерева Меркла в глубину

В подтверждении включения полный узел отправляет хеши, обозначенные как H_K и H_N на рис. 11.5, наряду с хешами, обозначенными там же как $H_{ABCDEFGH}$, H_{IJ}, H_L, H_M и H_{OP}. Местоположение хеша восстанавливается при обходе дерева Меркла в глубину, начиная с некоторых признаков. Процесс восстановления дерева, обозначенный пунктиром на рис 11.5, описывается далее.

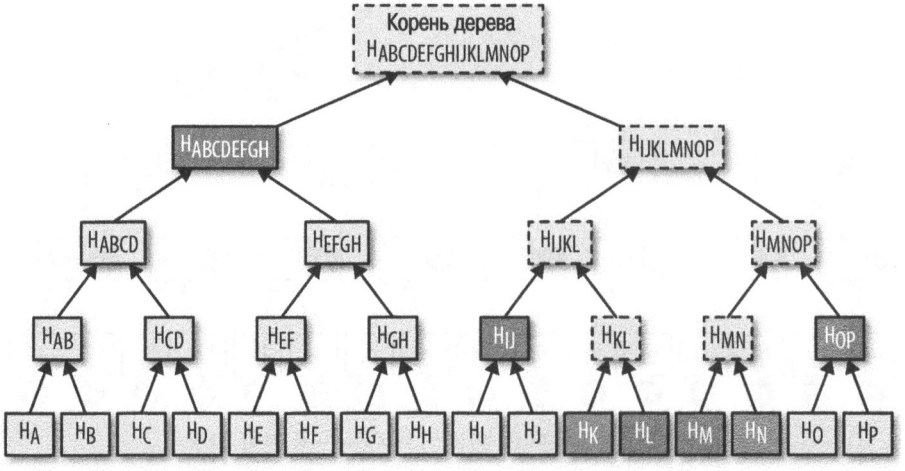

Рис. 11.5. Подтверждение включения в дереве Меркла

Структура дерева Меркла

Первым делом "тонкий" клиент создает общую структуру дерева Меркла. А поскольку деревья Меркла строятся снизу вверх от листьев к корню, "тонкому" клиенту потребуется количество существующих листьев, чтобы знать структуру этого дерева. Так, у дерева Меркла на рис. 11.5 имеется 16 листьев, поэтому "тонкий" клиент может создать пустое дерево Меркла следующим образом:

```
>>> import math
>>> total = 16
>>> max_depth = math.ceil(math.log(total, 2))     ❶
>>> merkle_tree = []     ❷
>>> for depth in range(max_depth + 1):     ❸
...     num_items = math.ceil(total / 2**(max_depth - depth))     ❹
...     level_hashes = [None] * num_items     ❺
...     merkle_tree.append(level_hashes)     ❻
>>> for level in merkle_tree:
...     print(level)
```

```
[None]
[None, None]
[None, None, None, None]
[None, None, None, None, None, None, None, None]
[None, None, None, None, None, None, None, None, None, None,\
None, None, None, None, None, None]
```

❶ На каждом уровне остается половина хешей, и поэтому логарифм $\log_2$ количества листьев определяет, сколько листьев имеется в дереве Меркла. Здесь это количество округляется с помощью метода `math.ceil()`, чтобы сократить его вдвое на каждом уровне. Но можно было бы поступить разумнее, употребив выражение `len(bin(total))-2`.

❷ В дереве Меркла корневой уровень будет храниться по нулевому индексу, а более низкий уровень — по единичному индексу и т.д. Иными словами, индекс обозначает "глубину" дерева, начиная с его вершины.

❸ В этом дереве Меркла имеются уровни от нулевого до `max_depth`, обозначая максимальную глубину данного дерева.

❹ На каждом последующем уровне количество узлов равно количеству всех узлов, деленному на 2, но с учетом текущей глубины дерева.

❺ Здесь любые еще неизвестные хеши задаются равными `None`.

❻ Следует иметь в виду, что `merkle_tree` — это список хешей в виде двухмерного массива.

Упражнение 5

Создайте пустое дерево Меркла с 27 узлами и выведите на экран каждый его уровень.

Программная реализация дерева Меркла

А теперь можно создать класс `MerkleTree`, в котором реализуется дерево Меркла, как показано ниже.

```
class MerkleTree:

  def __init__(self, total):
    self.total = total
    self.max_depth = math.ceil(math.log(self.total, 2))
    self.nodes = []
    for depth in range(self.max_depth + 1):
      num_items = math.ceil(
```

```
                self.total / 2**(self.max_depth - depth))
      level_hashes = [None] * num_items
      self.nodes.append(level_hashes)
    self.current_depth = 0     ❶
    self.current_index = 0

  def __repr__(self):     ❷
    result = []
    for depth, level in enumerate(self.nodes):
      items = []
      for index, h in enumerate(level):
        if h is None:
          short = 'None'
        else:
          short = '{}...'.format(h.hex()[:8])
        if depth == self.current_depth and index
                == self.current_index:
          items.append('*{}*'.format(short[:-2]))
        else:
          items.append('{}'.format(short))
      result.append(', '.join(items))
    return '\n'.join(result)
```

❶ Здесь сохраняется указатель на конкретный узел дерева, что может пригодиться в дальнейшем.

❷ Здесь выводится представление дерева.

Теперь, когда имеется пустое дерево Меркла, можно перейти к его заполнению, чтобы вычислить корень этого дерева. Если бы в каждом листе дерева Меркла находился отдельный хеш, то получить его корень можно было бы следующим образом:

```
>>> from merkleblock import MerkleTree
>>> from helper import merkle_parent_level
>>> hex_hashes = [
...   "9745f7173ef14ee4155722d1cbf13304339fd00d900b759c6f9d58579b5765fb",
...   "5573c8ede34936c29cdfdfe743f7f5fdfbd4f54ba0705259e62f39917065cb9b",
...   "82a02ecbb6623b4274dfcab82b336dc017a27136e08521091e443e62582e8f05",
...   "507ccae5ed9b340363a0e6d765af148be9cb1c8766ccc922f83e4ae681658308",
...   "a7a4aec28e7162e1e9ef33dfa30f0bc0526e6cf4b11a576f6c5de58593898330",
...   "bb6267664bd833fd9fc82582853ab144fece26b7a8a5bf328f8a059445b59add",
...   "ea6d7ac1ee77fbacee58fc717b990c4fcccf1b19af43103c090f601677fd8836",
...   "457743861de496c429912558a106b810b0507975a49773228aa788df40730d41",
...   "7688029288efc9e9a0011c960a6ed9e5466581abf3e3a6c26ee317461add619a",
...   "b1ae7f15836cb2286cdd4e2c37bf9bb7da0a2846d06867a429f654b2e7f383c9",
...   "9b74f89fa3f93e71ff2c241f32945d877281a6a50a6bf94adac002980aafe5ab",
```

```
...   "b3a92b5b255019bdaf754875633c2de9fec2ab03e6b8ce669d07cb5b18804638",
...   "b5c0b915312b9bdaedd2b86aa2d0f8feffc73a2d37668fd9010179261e25e263",
...   "c9d52c5cb1e557b92c84c52e7c4bfbce859408bedffc8a5560fd6e35e10b8800",
...   "c555bc5fc3bc096df0a0c9532f07640bfb76bfe4fc1ace214b8b228a1297a4c2",
...   "f9dbfafc3af3400954975da24eb325e326960a25b87fffe23eef3e7ed2fb610e",
... ]
>>> tree = MerkleTree(len(hex_hashes))
>>> tree.nodes[4] = [bytes.fromhex(h) for h in hex_hashes]
>>> tree.nodes[3] = merkle_parent_level(tree.nodes[4])
>>> tree.nodes[2] = merkle_parent_level(tree.nodes[3])
>>> tree.nodes[1] = merkle_parent_level(tree.nodes[2])
>>> tree.nodes[0] = merkle_parent_level(tree.nodes[1])
>>> print(tree)
*597c4baf.*
6382df3f..., 87cf8fa3...
3ba6c080..., 8e894862..., 7ab01bb6..., 3df760ac...
272945ec..., 9a38d037..., 4a64abd9..., ec7c95e1...,\
3b67006c..., 850683df..., d40d268b..., 8636b7a3...
9745f717..., 5573c8ed..., 82a02ecb..., 507ccae5...,\
a7a4aec2..., bb626766..., ea6d7ac1..., 45774386...,\
76880292..., b1ae7f15..., 9b74f89f..., b3a92b5b...,\
b5c0b915..., c9d52c5c..., c555bc5f..., f9dbfafc...
```

В приведенном выше коде заполняется дерево Меркла и получается его корень. Тем не менее в сообщении из сети могут быть предоставлены не все его листья, хотя оно могло бы содержать и некоторые внутренние узлы. Поэтому необходимо найти более разумный способ заполнения дерева Меркла.

И таким способом может стать *обход дерева*. В частности, дерево Меркла можно обойти в глубину, заполнив лишь те узлы, которые поддаются вычислению. Чтобы обойти дерево, необходимо отслеживать текущее положение в нем, пользуясь свойствами `self.current_depth` и `self.current_index`.

Для обхода дерева Меркла потребуются дополнительные методы, а также ряд других полезных методов, определяемых ниже.

```
class MerkleTree:
...
  def up(self):
    self.current_depth -= 1
    self.current_index //= 2

  def left(self):
    self.current_depth += 1
    self.current_index *= 2

  def right(self):
    self.current_depth += 1
```

```
    self.current_index = self.current_index * 2 + 1

  def root(self):
    return self.nodes[0][0]

  def set_current_node(self, value):     ❶
    self.nodes[self.current_depth][self.current_index] = value

  def get_current_node(self):
    return self.nodes[self.current_depth][self.current_index]

  def get_left_node(self):
    return self.nodes[self.current_depth + 1][self.current_index * 2]

  def get_right_node(self):
    return self.nodes[self.current_depth + 1][self.current_index * 2 + 1]

  def is_leaf(self):     ❷
    return self.current_depth == self.max_depth

  def right_exists(self):     ❸
    return len(self.nodes[self.current_depth + 1]) > \
              self.current_index * 2 + 1
```

❶ Здесь требуется возможность установить некоторое значение в текущем узле дерева.

❷ Здесь требуется выяснить, является ли текущий узел дерева листовым.

❸ В определенных случаях подходящего дочернего узла в дереве может и не оказаться. Ведь, обходя дерево, мы можем оказаться на уровне крайнего справа узла, на дочернем уровне которого находится нечетное количество элементов.

Итак, в нашем распоряжении имеются методы `left()`, `right()` и `up()` для обхода дерева Меркла. Поэтому воспользуемся этими методами, чтобы заполнить дерево, обходя его в глубину, как показано ниже.

```
>>> from merkleblock import MerkleTree
>>> from helper import merkle_parent
>>> hex_hashes = [
...   "9745f7173ef14ee4155722d1cbf13304339fd00d900b759c6f9d58579b5765fb",
...   "5573c8ede34936c29cdfdfe743f7f5fdfbd4f54ba0705259e62f39917065cb9b",
...   "82a02ecbb6623b4274dfcab82b336dc017a27136e08521091e443e62582e8f05",
...   "507ccae5ed9b340363a0e6d765af148be9cb1c8766ccc922f83e4ae681658308",
...   "a7a4aec28e7162e1e9ef33dfa30f0bc0526e6cf4b11a576f6c5de58593898330",
...   "bb6267664bd833fd9fc82582853ab144fece26b7a8a5bf328f8a059445b59add",
```

```
...    "ea6d7ac1ee77fbacee58fc717b990c4fcccf1b19af43103c090f601677fd8836",
...    "457743861de496c429912558a106b810b0507975a49773228aa788df40730d41",
...    "7688029288efc9e9a0011c960a6ed9e5466581abf3e3a6c26ee317461add619a",
...    "b1ae7f15836cb2286cdd4e2c37bf9bb7da0a2846d06867a429f654b2e7f383c9",
...    "9b74f89fa3f93e71ff2c241f32945d877281a6a50a6bf94adac002980aafe5ab",
...    "b3a92b5b255019bdaf754875633c2de9fec2ab03e6b8ce669d07cb5b18804638",
...    "b5c0b915312b9bdaedd2b86aa2d0f8feffc73a2d37668fd9010179261e25e263",
...    "c9d52c5cb1e557b92c84c52e7c4bfbce859408bedffc8a5560fd6e35e10b8800",
...    "c555bc5fc3bc096df0a0c9532f07640bfb76bfe4fc1ace214b8b228a1297a4c2",
...    "f9dbfafc3af3400954975da24eb325e326960a25b87fffe23eef3e7ed2fb610e",
... ]
>>> tree = MerkleTree(len(hex_hashes))
>>> tree.nodes[4] = [bytes.fromhex(h) for h in hex_hashes]
>>> while tree.root() is None:     ❶
...       if tree.is_leaf():     ❷
...         tree.up()
...       else:
...         left_hash = tree.get_left_node()
...         right_hash = tree.get_right_node()
...       if left_hash is None:     ❸
...         tree.left()
...       elif right_hash is None:     ❹
...         tree.right()
...       else:     ❺
...         tree.set_current_node(merkle_parent(left_hash, right_hash))
...         tree.up()
>>> print(tree)
597c4baf...
6382df3f..., 87cf8fa3...
3ba6c080..., 8e894862..., 7ab01bb6..., 3df760ac...
272945ec..., 9a38d037..., 4a64abd9..., ec7c95e1..., \
3b67006c..., 850683df..., d40d268b..., 8636b7a3...
9745f717..., 5573c8ed..., 82a02ecb..., 507ccae5..., \
a7a4aec2..., bb626766..., ea6d7ac1..., 45774386..., \
76880292..., b1ae7f15..., 9b74f89f..., b3a92b5b..., \
b5c0b915..., c9d52c5c..., c555bc5f..., f9dbfafc...
```

❶ Здесь обход дерева Меркла осуществляется до тех пор, пока не будет вычислен его корень. На каждом шаге цикла обходится конкретный узел дерева.

❷ Если это листовой узел, значит, искомый хеш достигнут, и поэтому остается лишь вернуться назад.

❸ Если отсутствует левый хеш, то сначала вычисляется его значение, а затем — хеш текущего узла.

❹ Если отсутствует правый хеш, то сначала вычисляется его значение, а затем — хеш текущего узла. Следует, однако, иметь в виду, что в данном случае уже имеется левый хеш, достигнутый в результате обхода в глубину.

❺ Поскольку уже имеются левый и правый хеши, вычисляется родительский хеш и устанавливается в текущем узле дерева Меркла. И как только оно будет установлено, можно вернуться назад.

Приведенный выше код будет действовать лишь в том случае, если количество листьев в дереве кратно степени двух, а крайние случаи, когда на отдельном уровне оказывается нечетное количество узлов, в данном коде во внимание не принимаются. Случай, когда родительский узел оказывается родительским для крайнего справа узла на уровне с нечетным количеством узлов, учитывается следующим образом:

```
>>> from merkleblock import MerkleTree
>>> from helper import merkle_parent
>>> hex_hashes = [
...   "9745f7173ef14ee4155722d1cbf13304339fd00d900b759c6f9d58579b5765fb",
...   "5573c8ede34936c29cdfdfe743f7f5fdfbd4f54ba0705259e62f39917065cb9b",
...   "82a02ecbb6623b4274dfcab82b336dc017a27136e08521091e443e62582e8f05",
...   "507ccae5ed9b340363a0e6d765af148be9cb1c8766ccc922f83e4ae681658308",
...   "a7a4aec28e7162e1e9ef33dfa30f0bc0526e6cf4b11a576f6c5de58593898330",
...   "bb6267664bd833fd9fc82582853ab144fece26b7a8a5bf328f8a059445b59add",
...   "ea6d7ac1ee77fbacee58fc717b990c4fcccf1b19af43103c090f601677fd8836",
...   "457743861de496c429912558a106b810b0507975a49773228aa788df40730d41",
...   "7688029288efc9e9a0011c960a6ed9e5466581abf3e3a6c26ee317461add619a",
...   "b1ae7f15836cb2286cdd4e2c37bf9bb7da0a2846d06867a429f654b2e7f383c9",
...   "9b74f89fa3f93e71ff2c241f32945d877281a6a50a6bf94adac002980aafe5ab",
...   "b3a92b5b255019bdaf754875633c2de9fec2ab03e6b8ce669d07cb5b18804638",
...   "b5c0b915312b9bdaedd2b86aa2d0f8feffc73a2d37668fd9010179261e25e263",
...   "c9d52c5cb1e557b92c84c52e7c4bfbce859408bedffc8a5560fd6e35e10b8800",
...   "c555bc5fc3bc096df0a0c9532f07640bfb76bfe4fc1ace214b8b228a1297a4c2",
...   "f9dbfafc3af3400954975da24eb325e326960a25b87fffe23eef3e7ed2fb610e",
...   "38faf8c811988dff0a7e6080b1771c97bcc0801c64d9068cffb85e6e7aacaf51",
... ]
>>> tree = MerkleTree(len(hex_hashes))
>>> tree.nodes[5] = [bytes.fromhex(h) for h in hex_hashes]
>>> while tree.root() is None:
...   if tree.is_leaf():
...     tree.up()
...   else:
...     left_hash = tree.get_left_node()
...     if left_hash is None:    ❶
...       tree.left()
...     elif tree.right_exists():    ❷
```

```
...         right_hash = tree.get_right_node()
...       if right_hash is None:     ❸
...         tree.right()
...       else:    ❹
...         tree.set_current_node(merkle_parent(
...                               left_hash, right_hash))
...         tree.up()
...     else:     ❺
...       tree.set_current_node(merkle_parent(left_hash, left_hash))
...       tree.up()
>>> print(tree)
0a313864...
597c4baf..., 6f8a8190...
6382df3f..., 87cf8fa3..., 5647f416...
3ba6c080..., 8e894862..., 7ab01bb6..., 3df760ac..., 28e93b98...
272945ec..., 9a38d037..., 4a64abd9..., ec7c95e1..., 3b67006c..., \
850683df..., d40d268b..., 8636b7a3..., ce26d40b...
9745f717..., 5573c8ed..., 82a02ecb..., 507ccae5..., a7a4aec2..., \
bb626766..., ea6d7ac1..., 45774386..., 76880292..., b1ae7f15..., \
9b74f89f..., b3a92b5b..., b5c0b915..., c9d52c5c..., c555bc5f..., \
f9dbfafc..., 38faf8c8...
```

❶ Если в данном узле отсутствует левый хеш, то обойти левый узел, поскольку все внутренние узлы гарантированно являются его левыми дочерними узлами.

❷ Сначала проверить, имеется ли у данного узла правый дочерний узел. Это условие истинно, если только данный узел не оказывается крайним справа, а на его дочернем уровне находится нечетное количество узлов.

❸ Если в данном узле отсутствует правый хеш, то обойти этот узел.

❹ Если в данном узле имеются левый и правый хеши, вычислить хеш в текущем узле с помощью метода `merkle_parent()`.

❺ Если в данном узле имеется левый хеш, но отсутствует правый хеш, значит, это крайний справа узел на данном уровне, поэтому объединить левый хеш дважды.

А теперь можно обойти в дереве количество листьев, которое не кратно степени двух.

Команда `merkleblock`

Полный узел, передающий древовидный блок Меркла, отправляет все сведения, требующиеся для того, чтобы проверить, находится ли пред-

ставляющая интерес транзакция в дереве Меркла. Для передачи именно этих сведений и служит сетевая команда `merkleblock`, как показано на рис. 11.6.

```
00000020df3b053dc46f162a9b00c7f0d5124e2676d47bbe7c5d0793a500000000000000ef445fef2
ed495c275892206ca533e7411907971013ab83e3b47bd0d692d14d4dc7c835b67d8001ac157e670bf
0d00000aba412a0d1480e370173072c9562becffe87aa661c1e4a6dbc305d38ec5dc088a7cf92e645
8aca7b32edae818f9c2c98c37e06bf72ae0ce80649a38655ee1e27d34d9421d940b16732f24b94023
e9d572a7f9ab8023434a4feb532d2adfc8c2c2158785d1bd04eb99df2e86c54bc13e1398628972174
00def5d72c280222c4cbaee7261831e1550dbb8fa82853e9fe506fc5fda3f7b919d8fe74b6282f927
63cef8e625f977af7c8619c32a369b832bc2d051ecd9c73c51e76370ceabd4f25097c256597fa898d
404ed53425de608ac6bfe426f6e2bb457f1c554866eb69dcb8d6bf6f880e9a59b3cd053e6c7060eea
caacf4dac6697dac20e4bd3f38a2ea2543d1ab7953e3430790a9f81e1c67f5b58c825acf46bd02848
384eebe9af917274cdfbb1a28a5d58a23a17977def0de10d644258d9c54f886d47d293a411cb62261
03b55635
```

- 00000020 - версия (4 байта в прямом порядке их следования)
- df3b...00 - предыдущий блок (32 байта в прямом порядке их следования)
- ef44...d4 - корень дерева Меркла (32 байта в прямом порядке их следования)
- dc7c835b - отметка времени (4 байта в прямом порядке их следования)
- 67d8001a - биты (4 байта)
- c157e670 - одноразовый номер (4 байта)
- bf0d0000 - общее количество транзакций (4 байта в прямом порядке их следования)
- 0a - number of hashes, varint
- ba41...61 - хеши (32 байта * количество хешей)
- 03b55635 - биты признаков

Рис. 11.6. Результат синтаксического анализа сетевой команды `merkleblock`

Первые шесть полей в данной команде точно такие же, как и в заголовке блока, описанном в главе 9. А в четырех последних полях хранится подтверждение включения.

В поле количества транзакций указывается, сколько листьев должно быть в данном конкретном дереве Меркла. Это дает "тонкому" клиенту возможность построить пустое дерево Меркла. В поле хешей содержатся хеши, обозначенные как $H_{ABCDEFGH}$, H_{IJ}, H_K, H_L, H_M, H_N и H_{OP} на рис. 11.5. А поскольку количество хешей, указываемое в поле хешей, не фиксировано, оно предваряется их количеством. И наконец, в поле признаков предоставляются сведения о местоположении хешей в дереве Меркла. Синтаксический анализ признаков осуществляется с помощью приведенного ниже метода `bytes_to_bits_field()`, в котором они преобразуются в список битов (т.е. единиц и нулей). Порядок следования байтов в данном случае не совсем обычен, но это означает, что биты признаков, требующиеся для восстановления корня дерева Меркла, преобразуются очень просто.

```
def bytes_to_bit_field(some_bytes):
  flag_bits = []
```

```
    for byte in some_bytes:
      for _ in range(8):
        flag_bits.append(byte & 1)
        byte >>= 1
    return flag_bits
```

Упражнение 6

Напишите метод `parse()` для класса `MerkleBlock`, в котором синтактически анализируется сетевая команда `merkleblock`.

Применение битов признаков и хешей

Биты признаков служат для обозначения хешей при обходе дерева Меркла в глубину. Ниже перечислены правила, по которым устанавливаются биты признаков.

1. Если значение хеша из данного узла задано в поле хешей (как обозначено $H_{ABCDEFGH}$, H_{IJ}, H_L, H_M, и H_{OP} на рис. 11.7), то в бите признака устанавливается 0.
2. Если узел является внутренним, а значение хеша в нем должно быть вычислено "тонким" клиентом, как выделено пунктиром на рис. 11.7, то в бите признака устанавливается 1.
3. Если узел является листовым и содержит представляющую интерес транзакцию (как обозначено H_K и H_N на рис. 11.7), то в бите признака устанавливается 1, а значение хеша в данном узле задается в поле хешей. Такие элементы оказываются включенными в дерево Меркла.

Учитывая структуру дерева Меркла, приведенную на рис. 11.7, можно утверждать следующее.

- Для корневого узла (1) в бите признака установлена 1, поскольку значение хеша в нем вычисляется "тонким" клиентом.
- Левый дочерний узел (2), обозначенный как $H_{ABCDEFGH}$, включается в состав поля хешей, и поэтому в бите признака установлен 0.
- Отсюда осуществляется переход к узлу (3), обозначенному как $H_{IJKLMNOP}$, вместо узлов, обозначенных как H_{ABCD} или H_{EFGH}, поскольку узел, обозначенный как $H_{ABCDEFGH}$, представляет оба эти узла, которые не нужны.

- Значение хеша вычисляется и в правом дочернем узле, обозначенном как $H_{IJKLMNOP}$, и поэтому в бите его признака установлена 1.

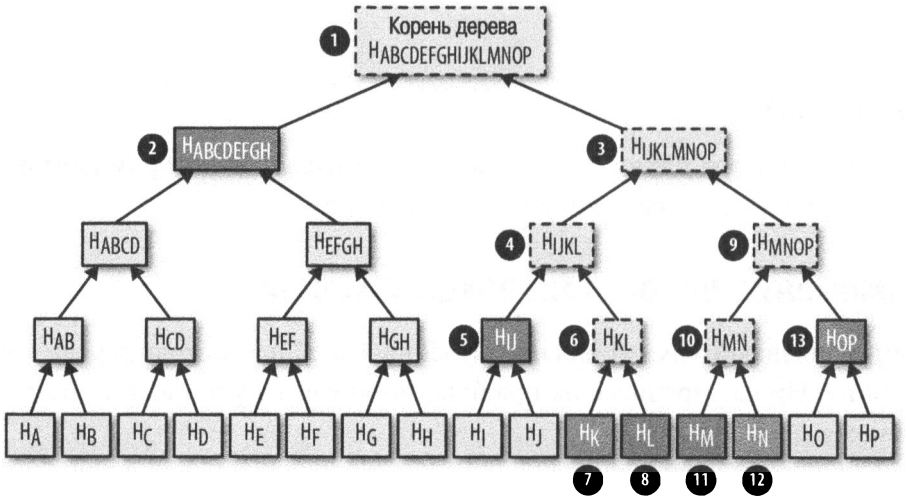

Рис. 11.7. Обработка древовидного блока Меркла

- Чтобы вычислить значение хеша в узле, обозначенном как $H_{IJKLMNOP}$, потребуются значения хешей для узлов (4) и (9), обозначенных как H_{IJKL} и H_{MNOP} соответственно. Следующим при обходе дерева в глубину является левый дочерний узел (4), обозначенный как H_{IJKL}. Именно с этого узла и продолжается обход рассматриваемого здесь дерева Меркла.
- Узел, обозначенный как H_{IJKL}, является внутренним, где вычисляется значение хеша, и поэтому в бите его признака установлена 1.
- Отсюда осуществляется переход к левому дочернему узлу (5), обозначенному как H_{IJ}. После возврата к этому узлу обход продолжится к узлу, обозначенному как H_{KL}.
- Следующим при обходе дерева в глубину является узел, обозначенный как H_{IJKL}. Его хеш включается в список хешей, а в бите его признака установлен 0.
- Узел, обозначенный как H_{KL}, является внутренним, где вычисляется значение хеша, и поэтому в бите его признака установлена 1.
- Узел (7), обозначенный как H_K, является листовым, а его присутствие в блоке подтверждается, и поэтому в бите его признака установлена 1.

- Узел (8), обозначенный как H_L, содержит значение хеша, включаемое в поле хешей, и поэтому в бите его признака установлен 0.
- Обход дерева продолжается вверх по дереву к узлу, обозначенному как H_{KL}, а затем вычисляется значение хеша в нем, поскольку содержимое узлов, обозначенных как H_K и H_L, уже известно.
- Обход дерева продолжается вверх по дереву к узлу, обозначенному как H_{IJKL}, а затем вычисляется значение хеша в нем, поскольку содержимое узлов, обозначенных как H_{IJ} и H_{KL}, уже известно.
- Обход дерева продолжается вверх по дереву к узлу, обозначенному как $H_{IJKLMNOP}$, но значение хеша вычислить в нем все же нельзя, поскольку еще не достигнут узел, обозначенный как H_{MNOP}.
- Обход дерева продолжается вниз к узлу, обозначенному как H_{MNOP} и являющемуся очередным внутренним узлом, поэтому в бите его признака установлена 1.
- Узел (10), обозначенный как H_{MN}, является очередным внутренним узлом, и поэтому в бите его признака установлена 1.
- Узел (11), обозначенный как H_M, содержит значение хеша, включенное в состав поля хешей, и поэтому в бите его признака установлен 0.
- Узел (12), обозначенный как H_N, представляет интерес, и поэтому в бите его признака установлена 1 и значение хеша в нем вводит в состав поля хешей.
- Обход дерева продолжается вверх к узлу, обозначенному как H_{MN}, где может быть теперь вычислено значение хеша.
- Обход дерева снова возвращается к узлу, обозначенному как H_{MNOP}, но значение хеша в нем нельзя вычислить, поскольку еще не достигнут узел, обозначенный как H_{OP}.
- Узел (13), обозначенный как H_{OP}, задан, и поэтому в бите его признака установлена 1, а значение хеша в нем является конечным в поле хешей.
- Обход дерева доходит до узла, обозначенного как H_{MNOP}, в котором может быть теперь вычислено значение хеша.
- Обход дерева доходит до узла, обозначенного как $H_{IJKLMNOP}$, в котором может быть теперь вычислено значение хеша.

- И наконец, обход дерева доходит до узла, обозначенного как $H_{ABCDEFGHIJKLMNOP}$ и являющегося корнем узла Меркла, который и вычисляется!

Таким образом, биты признаков для узлов (1–13) находятся в следующем состоянии:

```
1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0
```

Следовательно, семь хешей из проанализированного выше дерева Меркла содержатся в поле хешей в следующем порядке.

- $H_{ABCDEFGH}$
- H_{IJ}
- H_K
- H_L
- H_M
- H_N
- H_{OP}

Обратите внимание на буквы A–P, которыми выше обозначены хеши. Этих сведений достаточно, чтобы убедиться, что хеши H_K и H_N включены в блок.

Как следует из рис. 11.7, биты признаков устанавливаются по порядку обхода дерева в глубину. Всякий раз, когда в дереве Меркла достигается хеш (как, например, $H_{ABCDEFGH}$), его потомки пропускаются и обход дерева продолжается. Так, обход дерева продолжается от узла, обозначенного как $H_{ABCDEFGH}$, к узлу, обозначенному как $H_{IJKLMNOP}$, а не H_{ABCD}. Таким образом, биты признаков служат разумным механизмом для кодирования конкретных значений хешей в отдельных узлах дерева Меркла.

А теперь можно заполнить дерево Меркла и вычислить его корень при условии, что установлены надлежащие биты признаков и заданы соответствующие хеши, как показано ниже.

```
class MerkleTree:
...
def populate_tree(self, flag_bits, hashes):
  while self.root() is None:    ❶
    if self.is_leaf():    ❷
      flag_bits.pop(0)    ❸
      self.set_current_node(hashes.pop(0))    ❹
      self.up()
    else:
      left_hash = self.get_left_node()
      if left_hash is None:    ❺
        if flag_bits.pop(0) == 0:    ❻
          self.set_current_node(hashes.pop(0))
```

```
      self.up()
    else:
      self.left()     ❼
    elif self.right_exists():     ❽
      right_hash = self.get_right_node()
      if right_hash is None:     ❾
        self.right()
      else:     ❿
        self.set_current_node(merkle_parent(left_hash,
                              right_hash))
        self.up()
    else:     ⓫
      self.set_current_node(merkle_parent(left_hash, left_hash))
      self.up()
if len(hashes) != 0:     ⓬
  raise RuntimeError('hashes not all consumed {}'
                     .format(len(hashes)))
for flag_bit in flag_bits:     ⓭
  if flag_bit != 0:
    raise RuntimeError('flag bits not all consumed')
```

❶ Точка заполнения данного дерева Меркла служит для вычисления его корня. На каждом шаге цикла обрабатывается один узел дерева Меркла до тех пор, пока не будет вычислен его корень.

❷ Для листовых узлов хеш всегда задан.

❸ Вызов `flag_bits.pop(0)` служит в Python для извлечения бита следующего по очереди признака из соответствующего поля. Этот бит можно проанализировать для отслеживания представляющих интерес хешей, но здесь это пока что не делается.

❹ Вызов `hashes.pop(0)` служит для получения следующего по очереди хеша из поля хешей. Этот хеш необходимо установить в текущем узле дерева.

❺ Если в левом дочернем узле отсутствует значение хеша, то имеются две возможности: попытаться найти это значение в поле хешей или же вычислить его.

❻ Бит следующего признака указывает, следует или вычислить значение хеша в данном узле. Так, если в этом бите установлен 0, то искомое значение хеша извлекается как очередное из поля хешей. А если в этом бите установлена 1, то придется вычислить значение хеша для левого (а возможно, и правого) узла дерева.

❼ Это гарантированно левый дочерний узел дерева, и поэтому перейти к нему и получить находящееся в нем значение хеша.

❽ Здесь проверяется, существует ли правый узел.

❾ Здесь имеется левый, но не правый хеш, поэтому перейти к правому узлу и получить находящееся в нем значение хеша.

❿ Здесь имеются значения хешей из левого и правого узлов дерева, поэтому вычислить значение их родительского хеша как значение для текущего узла дерева.

⓫ Здесь имеется значение хеша из левого узла дерева, тогда как значение хеша в правом узле отсутствует. В таком случае вычислить значение его родительского хеша дважды по правилам, принятым для дерева Меркла.

⓬ Все хеши должны быть использованы, а иначе будут получены непригодные данные.

⓭ Все хеши должны быть использованы, а иначе будут получены непригодные данные.

Упражнение 7

Напишите метод `is_valid()` для класса `MerkleBlock`, чтобы реализовать в нем проверку древовидного блока Меркла на достоверность.

Заключение

Упрощенная проверка оплаты, конечно, удобна, но не лишена недостатков. И хотя полное описание упрощенной проверки оплаты выходит за рамки данной книги, а ее программная реализация не составляет большого труда, в большинстве "тонких" кошельков она все же не применяется; вместо этого доверительная информация получается из серверов поставщиков подобных кошельков. Главный недостаток упрощенной проверки оплаты заключается в том, что тем узлам сети, к которым вы подключаетесь, кое-что известно об интересующих вас транзакциях. А это означает, что, пользуясь упрощенной проверкой оплаты, вы частично теряете свою конфиденциальность. Подробнее об этом речь пойдет в главе 12, где рассматриваются фильтры Блума, через которые можно сообщить другим узлам сети о представляющих интерес транзакциях.

ГЛАВА 12

Фильтры Блума

В главе 11 пояснялось, как проверять древовидный блок Меркла на достоверность. Полный узел может предоставить подтверждение включения для представляющих интерес транзакций по сетевой команде `merkleblock`. Но откуда полному узлу известно, какие именно транзакции представляют интерес?

"Тонкий" клиент мог бы сообщить полному узлу свои адреса (или открытые ключи из сценария ScriptPubKey). Полный узел может проверить транзакции, соответствующие этим адресам, но в то же время это может нарушить конфиденциальность "тонкого" клиента. Ведь "тонкому" клиенту вряд ли захочется раскрывать полному узлу, что он, например, обладает суммой 1000 биткойнов. Утечки конфиденциальности означают утечку информации, и в биткойне, как правило, рекомендуется при всякой возможности избегать любых утечек конфиденциальности.

Но в одном случае "тонкий" клиент может сообщить полному узлу достаточно сведений, чтобы создать *надмножество* всех представляющих интерес транзакций. Чтобы создать такое надмножество, придется воспользоваться так называемым *фильтром Блума*.

Что такое фильтр Блума

Фильтр Блума служит фильтром для всех возможных транзакций. Полные узлы выполняют транзакции через фильтр Блума и отсылают команды `merkleblock` для выполнения пропускаемых через него транзакций.

Допустим, что всего имеется 50 транзакций, а "тонкого" клиента интересует лишь одна из них. В частности, "тонкому" клиенту требуется "скрыть" транзакцию из группы, состоящей из пяти транзакций. Для этого требуется функция, составляющая 50 транзакций в 10 разных групп, чтобы полный узел смог затем отправить, условно говоря, единую группу транзакций. Такое

группирование было бы *детерминированным*, т.е. оно должно быть постоянно одинаковым. Как же этого добиться? Решение в том, чтобы получить с помощью хеш-функции детерминированное число и остаток по модулю для организации транзакций в группы.

Фильтр Блума является структурой вычислительной техники, которую можно применить к любым данным из множества. Допустим, что имеется один элемент вроде строки "HelloWorld" (Здравствуй, мир), для которого требуется создать фильтр Блума. Для этого потребуется хеш-функция, и поэтому воспользуемся уже знакомой нам хеш-функцией hash256. Ниже показано, каким образом процесс выяснения, к какой именно группе относится данный элемент, реализуется непосредственно в коде.

```
>>> from helper import hash256
>>> bit_field_size = 10    ❶
>>> bit_field = [0] * bit_field_size
>>> h = hash256(b'hello world')    ❷
>>> bit = int.from_bytes(h, 'big') % bit_field_size    ❸
>>> bit_field[bit] = 1    ❹
>>> print(bit_field)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

❶ В переменной `bit_field` сохраняется список "групп" из 10 элементов.

❷ Здесь исходный элемент хешируется с помощью хеш-функции hash256.

❸ Здесь полученный результат интерпретируется как целочисленное значение, представленное байтами в обратном порядке их следования, а также остатка по модулю 10, чтобы определить группу, к которой относится данный элемент.

❹ Здесь обозначается группа, которая требуется в фильтре Блума.

Все, что было сделано в приведенном выше коде, схематически показано на рис. 12.1.

Таким образом, рассматриваемый здесь фильтр Блума состоит из следующих элементов.

1. Размер битового поля.
2. Применяемая хеш-функция, а также алгоритм преобразования в число.
3. Битовое поле, обозначающее представляющую интерес группу.

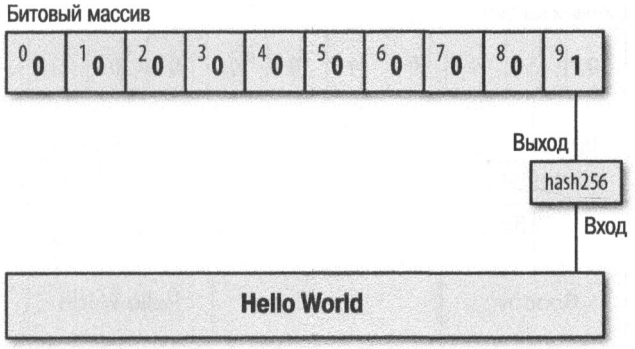

Рис. 12.1. 10-разрядный фильтр Блума с одним элементом

Такой фильтр пригоден для единственного элемента, поэтому он подходит и для представляющего интерес адреса, сценария ScriptPubKey или идентификатора транзакции. А что делать, если нас интересует не один элемент?

Выполнить второй элемент можно через тот же самый фильтр, а также установить 1 в соответствующем бите. И тогда полный узел может отправить несколько групп транзакций вместо одной группы. Итак, создадим фильтр Блума с двумя элементами ("Hello world" и "Goodbye"), воспользовавшись следующим кодом:

```
>>> from helper import hash256
>>> bit_field_size = 10
>>> bit_field = [0] * bit_field_size
>>> for item in (b'hello world', b'goodbye'):     ❶
...   h = hash256(item)
...   bit = int.from_bytes(h, 'big') % bit_field_size
...   bit_field[bit] = 1
>>> print(bit_field)
[0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
```

❶ Здесь создается фильтр для двух элементов, хотя данную процедуру можно расширить и на большее количество элементов.

Порядок создания фильтра Блума с двумя элементами схематически показан на рис. 12.2.

Если количество всех возможных элементов равно 50, то через такой фильтр в среднем будет пропущено 10, а не 5 представляющих интерес элементов, как в том случае, если бы он был одноэлементным. Ведь в данном случае возвращаются две группы элементов вместо одной.

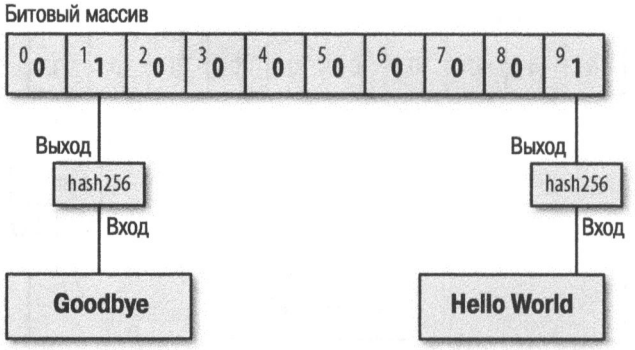

Рис. 12.2. 10-разрядный фильтр Блума с двумя элементами

Упражнение 1

Рассчитайте фильтр Блума для строк "Hello world" и "Goodbye", применив хеш-функцию hash160 к битовому полю размером 10.

Продвижение на шаг дальше

Допустим, что пространство всех элементов составляет 1 миллион и требуется оставить размер группы равным 5. И для этого потребуется фильтр Блума длиной 1000000 / 5 = 200000 битов. При этом каждая группа будет в среднем состоять из 5 элементов, и тогда мы получим в 5 раз больше элементов, чем нужно, т.е. лишь 20% от всего количества элементов будут представлять для нас интерес. Но передать данные объемом 200000 битов или 25000 байтов не так-то просто. Можно ли в таком случае поступить лучше?

Если воспользоваться несколькими хеш-функциями в фильтре Блума, то можно значительно сократить размер битового поля. Так, если применить 5 хеш-функций к битовому полю размером 32, то в итоге получится 32!/(27!5!) ~ 200000 возможных комбинаций из 5 битов в этом 32-разрядном поле. А из 1 миллиона возможных элементов в среднем 5 должны иметь такую комбинацию из 5 разрядов. Таким образом, вместо 25 Кбайтов в данном случае можно передать лишь 32 бита или 4 байта!

Ниже показано, каким образом такой фильтр реализуется непосредственно в коде. Для простоты в данном примере используются то же самое 10-разрядное поле и два представляющих интерес элемента.

```
>>> from helper import hash256, hash160
>>> bit_field_size = 10
```

```
>>> bit_field = [0] * bit_field_size
>>> for item in (b'hello world', b'goodbye'):
...     for hash_function in (hash256, hash160):     ❶
...         h = hash_function(item)
...         bit = int.from_bytes(h, 'big') % bit_field_size
...         bit_field[bit] = 1
>>> print(bit_field)
[1, 1, 1, 0, 0, 0, 0, 0, 0, 1]
```

❶ Здесь циклически выполняются две разные хеш-функции (hash256 и hash160), но с тем же успехом можно сделать больше.

Порядок создания фильтра Блума с двумя элементами и хеш-функциями схематически показан на рис. 12.3.

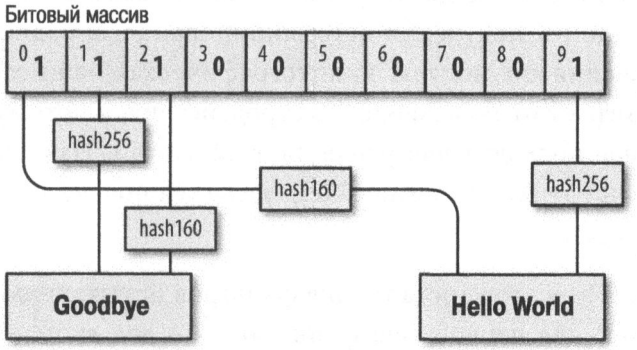

Рис. 12.3. 10-разрядный фильтр Блума с двумя элементами и хеш-функциями

Фильтр Блума можно оптимизировать, изменив количество хеш-функций и размер битогового поля, чтобы получить требуемую долю ложноположительных результатов.

Фильтры Блума по протоколу BIP0037

В протоколе BIP0037 определяются фильтры Блума, применяемые при передаче данных по сети. Ниже перечислены сведения, хранящиеся в фильтре Блума.

1. Размер битового поля или количество имеющихся групп. Этот размер указывается в байтах (по 8 битов на каждый байт) и округляется по мере необходимости.
2. Количество хеш-функций.

3. "Настройка", способная немного изменить фильтр Блума, если он встретит слишком много элементов.
4. Битовое поле, получаемое в результате выполнения фильтра Блума над представляющими интерес элементами.

Несмотря на возможность определить немало хеш-функций (sha512, keccak384, ripemd160, blake256 и т.д.), на практике применяется единственная хеш-функция, но с разными начальными случайными значениями. Благодаря этому упрощается реализация данного фильтра.

Мы будем пользоваться здесь хеш-функцией, называемой *murmur3*. В отличие от хеш-функции sha256, хеш-функция murmur3 не является криптографически безопасной, но действует намного быстрее. И хотя для решения задачи фильтрации и получения детерминированного, равномерно распределенного значения по модулю криптографическая защита не требуется, оно только выиграет от повышения быстродействия, а потому хеш-функция murmur3 больше подходит для решения этой задачи. Начальное случайное значение для хеш-функции murmur3 определяется по следующей формуле:

```
i*0xfba4c795 + tweak
```

Здесь `fba4c795` — это константа для фильтров Блума, применяемых в биткойне, `i` равно 0 для первой хеш-функции, 1 — для второй хеш-функции, 2 — для третьей хеш-функции и так далее, а `tweak` — доля энтропии, которая может быть введена, если не удовлетворяют результаты одной настройки. Хеш-функции и размер битового поля служат для вычисления содержимого битового поля, которое затем передается, как показано ниже.

```
>>> from helper import murmur3    ❶
>>> from bloomfilter import BIP37_CONSTANT    ❷
>>> field_size = 2
>>> num_functions = 2
>>> tweak = 42
>>> bit_field_size = field_size * 8
>>> bit_field = [0] * bit_field_size
>>> for phrase in (b'hello world', b'goodbye'):    ❸
...     for i in range(num_functions):    ❹
...         seed = i * BIP37_CONSTANT + tweak    ❺
...         h = murmur3(phrase, seed=seed)    ❻
...         bit = h % bit_field_size
...         bit_field[bit] = 1
>>> print(bit_field)
[0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

❶ Хеш-функция murmur3 реализуется в исходном файле `helper.py` только на языке Python.

❷ Константа `BIP37_CONSTANT` содержит число `fba4c795`, указанное в протоколе BIP0037.

❸ Здесь циклически перебираются представляющие интерес элементы.

❹ Здесь применяются две хеш-функции.

❺ Здесь реализуется формула для определения начального случайного значения.

❻ Хеш-функция murmur3 возвращает случайное число, и поэтому его не нужно преобразовывать в целое число.

В четырех из шестнадцати битов (т.е. двух байтов) реализованного выше фильтра Блума установлена 1, и поэтому вероятность прохождения произвольного элемента через этот фильтр равна 1/4 × 1/4 = 1/16. Так, если количество всех элементов равно 160, клиент получит в среднем 10 элементов, 2 из которых будут представлять интерес.

А теперь можно приступить к определению класса `BloomFilter`, реализующего фильтр Блума следующим образом:

```
class BloomFilter:

  def __init__(self, size, function_count, tweak):
    self.size = size
    self.bit_field = [0] * (size * 8)
    self.function_count = function_count
    self.tweak = tweak
```

Упражнение 2

Если задан фильтр Блума с параметрами `size=10`, `function_count=5`, `tweak=99`, то какие байты устанавливаются после ввода приведенных ниже элементов? (*Подсказка:* воспользуйтесь функцией `bit_field_to_bytes()` из исходного файла `helper.py` для преобразования содержимого битового поля в байты.)

- `b'Hello World'`
- `b'Goodbye!'`

Упражнение 3

Напишите метод `add()` для класса `BloomFilter`, чтобы реализовать в нем ввод элементов в фильтр Блума.

Загрузка фильтра Блума

Как только "тонкий" клиент создаст фильтр Блума, ему придется уведомить полный узел об этом фильтре, чтобы полный узел смог отправлять подтверждения включения. Для этого "тонкий" клиент должен, прежде всего, установить 0 в дополнительном признаке пересылки из сообщения о версии (см. главу 10). Этим полному узлу указывается не отправлять сообщения о транзакциях, если только они не пройдут фильтр Блума или не будут запрошены специально. После установки признака пересылки "тонкий" клиент должен передать полному узлу сам фильтр Блума. Для этого и служит команда `filterload`, структура которой приведена на рис. 12.4.

```
0a4000600a08000001094005000000630000000

- 0a4000600a080000010940 - Битовое поле переменной длины
- 05000000 - Подсчет хешей, 4 байта в прямом порядке их следования
- 63000000 - Настройка, 4 байта в прямом порядке их следования
- 00 - Признак совпадающего элемента
```

Рис. 12.4. Результат синтаксического анализа команды `filterload`

Элементы фильтра Блума кодируются в байтах. Битовое поле, поля подсчета хеш-функций и настройки кодируются в сообщении о фильтре Блума. А последнее поле с признаком совпадающего элемента служит для запроса полного узла на ввод любых совпадающих транзакций в фильтр Блума.

Упражнение 4

Напишите метод `filterload()` для класса `BloomFilter`, чтобы реализовать в нем загрузку фильтра Блума.

Получение древовидных блоков Меркла

"Тонкому" клиенту потребуется еще одна команда, чтобы получить из полного узла сведения об интересующих его транзакциях, находящихся в древовидном блоке из дерева Меркла. Сведения о блоках и транзакциях передает команда `getdata`, а конкретный тип данных, которые потребуются "тонкому"

клиенту из полного узла, называется *фильтрованным блоком*. Такой блок является запросом транзакций, проходящим через фильтр Блума в форме древовидного блока Меркла. Иными словами, "тонкий" клиент может запросить древовидные блоки Меркла, в которых находятся транзакции, которые интересуют данного клиента и совпадают с критерием прохождения через фильтр Блума. Структура команды `getdata` приведена на рис. 12.5.

```
020300000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
0000000000003000000104984793958 5b0652fba793661c361223446b6fc410
89b8be00000000000000

        - 02 - Количество элементов данных
        - 03000000 - Тип элемента данных (транзакция, блок, фильтрованный блок,
              компактный блок), в прямом порядке следования байтов
        - 30...00 - Идентификатор хеша
```

Рис. 12.5. Результат синтаксического анализа команды `getdata`

Количество элементов данных типа `varint` обозначает, сколько требуется таких элементов. Каждый элемент данных относится к конкретному типу. В частности, значение типа 1 относится к транзакции (см. главу 5), типа 2 — к обычному блоку (см. главу 9), типа 3 — к древовидному блоку Меркла (см. главу 11), а типа 4 — к компактному блоку (этот тип в данной книге не рассматривается).

В исходном файле `network.py` можно создать следующее сообщение:

```
class GetDataMessage:
  command = b'getdata'

  def __init__(self):
    self.data = []     ❶

  def add_data(self, data_type, identifier):
    self.data.append((data_type, identifier))     ❷
```

❶ Сохранить требующиеся элементы данных.

❷ Ввести элементы данных в сообщение, используя метод `add_data()`.

Упражнение 5

Напишите метод `serialize()` для класса `GetDataMessage`, чтобы реализовать в нем сериализацию получаемых элементов данных.

Получение представляющей интерес транзакции

"Тонкий" клиент, загружающий фильтр Блума из полного узла, получит в свое распоряжение все необходимые сведения, позволяющие ему убедиться, что интересующие его транзакции действительно входят в состав конкретных блоков, как показано ниже.

```
>>> from bloomfilter import BloomFilter
>>> from helper import decode_base58
>>> from merkleblock import MerkleBlock
>>> from network import FILTERED_BLOCK_DATA_TYPE, GetHeadersMessage,\
GetDataMessage, HeadersMessage, SimpleNode
>>> from tx import Tx
>>> last_block_hex = '00000000000538d5c2246336644f9a4956551afb44ba\
47278759ec55ea912e19'
>>> address = 'mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv'
>>> h160 = decode_base58(address)
>>> node = SimpleNode('testnet.programmingbitcoin.com', testnet=True,\
logging= False)
>>> bf = BloomFilter(size=30, function_count=5, tweak=90210)     ❶
>>> bf.add(h160)     ❷
>>> node.handshake()
>>> node.send(bf.filterload())     ❸
>>> start_block = bytes.fromhex(last_block_hex)
>>> getheaders = GetHeadersMessage(start_block=start_block)
>>> node.send(getheaders)     ❹
>>> headers = node.wait_for(HeadersMessage)
>>> getdata = GetDataMessage()     ❺
>>> for b in headers.blocks:
...     if not b.check_pow():
...       raise RuntimeError('proof of work is invalid')
...     getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())     ❻
>>> node.send(getdata)     ❼
>>> found = False
>>> while not found:
...     message = node.wait_for(MerkleBlock, Tx)     ❽
...     if message.command == b'merkleblock':
...       if not message.is_valid():     ❾
...         raise RuntimeError('invalid merkle proof')
...       else:
...         for i, tx_out in enumerate(message.tx_outs):
...           if tx_out.script_pubkey.address(testnet=True) \
...                   == address:     ❿
...             print('found: {}:{}'.format(message.id(), i))
...             found = True
...             break
```

```
found: e3930e1e566ca9b75d53b0eb9acb7607f547e1182d1d22bd4b661\
cfe18dcddf1:0
```

❶ Создать фильтр Блума размером 30 байтов, использующий хеш-функции и настройку, особенно распространенную в 1990-е годы.

❷ Отфильтровать по приведенному выше адресу.

❸ Отправить команду `filterload` из созданного фильтра Блума.

❹ Получить заголовки блоков после шестнадцатеричного идентификатора последнего блока (`last_block_hex`).

❺ Создать сообщение для получения данных для древовидных блоков Меркла, в которых могут присутствовать представляющие интерес транзакции.

❻ Запросить древовидный блок Меркла, чтобы убедиться в наличии в нем представляющих интерес транзакций. В большинстве блоков они, вероятнее всего, будут отсутствовать.

❼ Запросить в сообщении для получения данных 2000 древовидных блоков Меркла после определения блока по его шестнадцатеричному идентификатору (`last_block_hex`).

❽ Ожидать команды `merkleblock`, подтверждающей включение, а также команды `tx`, предоставляющей искомую транзакцию.

❾ Проверить, подтверждает ли древовидный блок Меркла включение транзакции.

❿ Найти неизрасходованные выводы транзакций UTXO по конкретному адресу (`address`) и вывести найденное на экран.

В данном случае проанализированы 2000 блоков после конкретного блока, чтобы обнаружить неизрасходованные выводы транзакций UTXO, соответствующие конкретному адресу. А поскольку все сделано безо всякой помощи обозревателя блоков, в какой-то степени это сохранило нашу конфиденциальность.

Упражнение 6

Получите идентификатор текущего блока в сети testnet, отправьте себе немного монет по сети testnet, найдите неизрасходованный вывод транзакции UTXO, соответствующий этим монетам, *не* пользуясь обозревателем блоков,

создайте транзакцию, используя данный UTXO в качестве ввода, а также перешлите сообщение `tx` по сети testnet.

Заключение

В этой главе было показано, как создать все, что требуется для подключения "тонкого" клиента по одноранговой сети, запроса и получения неизрасходованных выводов транзакции UTXO, необходимых для построения транзакции. И все это можно сделать, сохранив конфиденциальность с помощью фильтра Блума. А теперь перейдем к протоколу Segwit, определяющему новый тип транзакции, внедренный в 2017 году.

ГЛАВА 13

Протокол Segwit

Название *Segwit* (англ.) означает "раздельное заверение" и служит для обратной совместимости или нерадикального изменения в протоколе криптовалюты (так называемой "мягкой вилки"), внедренное в сети биткойна в августе 2017 года. Несмотря на всю противоречивость внедрения данной технологии, ее функциональные возможности требуют некоторых пояснений. В этой главе описываются принцип действия протокола Segwit, причины его обратной совместимости и что, собственно, он активизирует.

Если кратко, то в Segwit были внедрены многие изменения, в том числе следующие.

- Увеличение размера блока
- Устранение податливости транзакций
- Контроль версий Segwit, чтобы прояснить пути для обновления
- Исправление алгоритма квадратичного хеширования
- Обеспечение безопасности вычисления оплаты из электронного кошелька в автономном режиме

Если не рассмотреть особенности реализации протокола Segwit, то его назначение будет не вполне очевидным. Поэтому начнем с рассмотрения элементарного типа транзакции Segwit: оплаты по хешу открытого ключа с раздельным заверением.

Оплата по хешу открытого ключа с раздельным заверением

Оплата по хешу открытого ключа с раздельным заверением (p2wpkh) является одним из четырех сценариев, определенных по технологии Segwit в протоколах BIP0141 и BIP0143. Этот умный контракт во многом действует

подобно оплате по хешу открытого ключа, поэтому он был и назван сходным образом. Главное отличие от сценария p2pkh состоит в том, что данные для сценария ScriptSig теперь находятся в поле заверения. Такая реорганизация сделана для устранения податливости транзакции.

Податливость транзакции

Податливость транзакции (transaction malleability) — это способность изменять идентификатор транзакции, не меняя ее назначение. Исполнительный директор биржи криптовалюты Mt. Gox Марк Карпелес назвал податливость транзакции причиной, по которой на этой бирже не разрешались операции по изъятию средств в 2013 году.

Податливость идентификатора служит важным соображением при создании платежных каналов, которые являются атомарными единицами платежного протокола Lightning Network. Идентификатор податливой транзакции намного усложняет безопасное создание транзакций в платежных каналах.

Трудности, обусловленные податливостью транзакций, объясняются тем, что идентификатор (т.е. хеш-код hash256) транзакции вычисляется по всей транзакции в целом. Большинство полей в транзакции нельзя изменить, не признав недостоверной подпись данной транзакции, а следовательно, и саму транзакцию, поэтому такие поля не вызывают особых затруднений в отношении податливости, а по существу, уязвимости транзакций.

К числу полей, которые допускают какое-то манипулирование без признания подписи недействительной, относится поле ScriptSig на каждом вводе транзакции. Поле ScriptSig опорожняется перед созданием хеша подписи (см. главу 7), и поэтому его содержимое можно изменить, не нарушив достоверность подписи. Как пояснялось в главе 3, подписи содержат случайную составляющую, а это означает, что два разных поля ScriptSig могут, по существу, означать одно и то же, хотя и различаться побайтно.

Вследствие этого поле ScriptSig оказывается *податливым*, т.е. его можно изменить, не меняя его назначение. Это означает, что идентификатор транзакции и вся транзакция в целом оказываются податливыми. Податливость идентификатора транзакции означает, что любые *зависимые* транзакции, к которым относится любая транзакция, расходующая один из податливых выводов транзакции, не могут быть созданы подобным образом, чтобы гарантировать ее достоверность. Хеш предыдущей транзакции не определен,

а следовательно, гарантировать достоверность содержимого поля ввода зависимой транзакции нельзя.

Как правило, это не вызывает особых затруднений. Ведь как только транзакция входит в цепочку блоков, ее идентификатор остается фиксированным и больше не поддается изменениям, по крайней мере не требует поиска подтверждения работы! Но в платежных каналах зависимые транзакции создаются *прежде*, чем фондирующая транзакция вводится в цепочку блоков.

Устранение податливости транзакций

Чтобы устранить податливость транзакции, достаточно опорожнить поле ScriptSig и разместить данные в другом поле, которое не применяется для вычисления идентификатора транзакции. В сценарии p2wpkh подпись и открытый ключ являются элементами, извлекаемыми из поля ScriptSig, а следовательно, их можно переместить в поле заверения, которое не применяется для вычисления идентификатора транзакции. Таким образом, идентификатор транзакции остается устойчивым по мере исчезновения вектора податливости. Поле заверения и результат сериализации всей транзакции по протоколу Segwit отправляются только тем узлам, которые их запрашивают. Иными словами, прежние узлы, которые не перешли на технологию Segwit, не принимают поле заверения и не проверяют достоверность открытого ключа и подписи.

Это кажется уже знакомым, и так оно и есть. Ведь это похоже на сценарий p2sh (см. главу 8) в том отношении, что новые узлы выполняют дополнительную проверку достоверности по сравнению со старыми узлами. Именно поэтому Segwit и называется "мягкой вилкой" (т.е. обратно совместимым обновлением протокола), а не "жесткой вилкой" (т.е. обратно несовместимым обновлением протокола).

Транзакции по сценарию p2wpkh

Чтобы стал понятнее принцип действия Segwit, полезно показать, как выглядит транзакция, когда она отсылается старому узлу (рис. 13.1), в отличие от нового узла (рис. 13.2).

Различие результатов сериализации, приведенных на рис. 13.1 и 13.2, состоит в том, что во втором случае транзакция, сериализированная по протоколу Segwit, содержит поля маркера, признака и заверения, а в остальном обе транзакции одинаковы. Идентификатор транзакции оказывается

неподатливым потому, что в первом случае сериализация служит для вычисления идентификатора транзакции.

```
01000000011 5e180dc28a2327e687facc33f10f2a20da717e5548406f7ae8b4c811072f8560100000
000ffffffff0100b4f50500000001976a9141d7cd6c75c2e86f4cbf98eaed221b30bd9a0b92888ac
00000000

        - 01000000 - версия
        - 01 - # количество входов
        - 15e1...56 - хеш предыдущей транзакции
        - 01000000 - индекс предыдущей транзакции
        - 00 - ScriptSig
        - ffffffff - последовательность
        - 01 - # количество выходов
        - 00b4...00 - суммы на выходах
        - 1976...ac - ScriptPubKey
        - 00000000 - время блокировки
```

Рис. 13.1. Оплата по хешу открытого ключа с раздельным заверением (p2wpkh) до внедрения программного обеспечения по протоколу BIP0141

```
01000000 00 010115e180dc28a2327e687facc33f10f2a20da717e5548406f7ae8b4c811072f856010
0000000ffffffff0100b4f505000000001976a9141d7cd6c75c2e86f4cbf98eaed221b30bd9a0b928
88ac02483045022100df7b7e5cda14ddf91290e02ea10786e03eb11ee36ec02dd862fe9a326bbcb7f
d02203f5b4496b667e6e281cc654a2da9e4f08660c620a1051337fa8965f727eb19190121038262a6
c6cec93c2d3ecd6c6072efea86d02ff8e3328bbd0242b20af3425990ac00000000

        - 01000000 - версия
        - 00 - маркер Segwit
        - 01 - признак Segwit
        - 01 - # количество входов
        - 15e1...56 - хеш предыдущей транзакции
        - 01000000 - индекс предыдущей транзакции
        - 00 - ScriptSig
        - ffffffff - последовательность
        - 01 - # количество выходов
        - 00b4...00 - суммы на выходах
        - 1976...ac - ScriptPubKey
        - 0248...61 - заверение
        - 00000000 - время блокировки
```

Рис. 13.2. Оплата по хешу открытого ключа с раздельным заверением (p2wpkh) после внедрения программного обеспечения по протоколу BIP0141

Поле заверения в p2wpkh содержит подпись и открытый ключ в качестве двух его элементов. Это поле служит для проверки достоверности только обновленных узлов. Поле ScriptPubKey для p2wpkh содержит последовательность команд `OP_0 <20-байтовый хеш>`, поле ScriptSig пусто, как следует из рис. 13.1 и 13.2, а объединенный сценарий приведен на рис. 13.3.

Рис. 13.3. Поля ScriptPubKey и ScriptSig для оплаты по хешу открытого ключа с раздельным заверением (p2wpkh)

Обработка объединенного сценария начинается так, как показано на рис. 13.4.

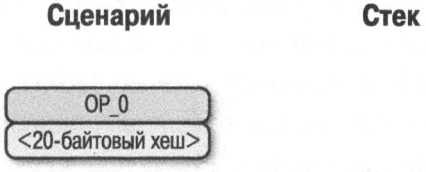

Рис. 13.4. Начальное состояние сценария p2wpkh

Операция `OP_0` размещает `0` в стеке (рис. 13.5).

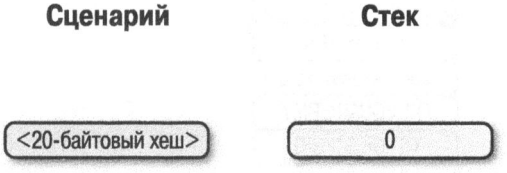

Рис. 13.5. Первая стадия выполнения сценария p2wpkh

20-байтовый хеш является элементом и поэтому размещается в стеке (рис. 13.6).

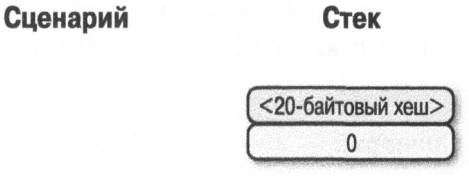

Рис. 13.6. Вторая стадия выполнения сценария p2wpkh

На данной стадии старые узлы остановятся, потому что в стеке больше не остается команд Script для обработки. А поскольку на вершине стека

находится ненулевой элемент, он будет трактоваться как достоверный сценарий. И это очень похоже на p2sh в том отношении, что старые узлы больше не смогут выполнять проверку на достоверность, а новые узлы будут действовать по особому правилу Segwit, во многом подобному особому правилу для p2sh. Как упоминалось в главе 8, конкретная последовательность команд сценария `<RedeemScript>` `OP_HASH160` `<хеш>` `OP_EQUAL` приводит в действие особое правило для p2sh.

А в p2wpkh последовательность команд сценария такова: `OP_0` `<20-байтовый хеш>`. Когда в сценарии встречается такая последовательность команд, открытый ключ и подпись из поля заверения и 20-байтовый хеш вводятся в набор команд именно в такой последовательности, как и в p2pkh, а именно: `<подпись>` `<открытый ключ>` `OP_DUP` `OP_HASH160` `<20-байтовый хеш>` `OP_EQUALVERIFY` `OP_CHECKSIG`. Состояние выполнения сценария p2wpkh на следующей стадии приведено на рис. 13.7.

Рис. 13.7. Третья стадия выполнения сценария p2wpkh

Остальная часть обработки сценария p2wpkh выполняется таким же образом, как и сценария p2pkh (см. главу 6). А его конечное состояние отличается тем, что в стеке остается только 1, но лишь в том случае, если 20-байтовый хеш представляет собой хеш-код hash160 открытого ключа, а подпись действительна (рис. 13.8).

Рис. 13.8. Четвертая стадия выполнения сценария p2wpkh

В старом узле обработка останавливается, если *<20-байтовый хеш>* равен нулю, поскольку старым узлам неизвестно особое правило Segwit. И лишь в обновленных узлах будет выполнена остальная часть проверки достоверности, как и в p2sh. Следует, однако, иметь в виду, что объем данных, которыми обмениваются обновленные узлы, оказывается меньше, чем объем старых узлов. Кроме того, узлам предоставляется возможность не загружать, а следовательно, и не проверять транзакции, если они устарели на *x* блоков. По существу, подпись заверена многими лицами, и поэтому узел может довериться им, посчитав ее действительной вместо того, чтобы проверить ее достоверность непосредственно.

Следует также иметь в виду, что данное особое правило относится к версии 0 протокола Segwit, а в версии Segwit 1 процедура обработки может быть совсем другой. В частности, *<20-байтовый хеш>* 1 может быть особой сценарной последовательностью, приводящей в действие другое правило. В последующих обновлениях Segwit могут быть внедрены подписи Шнорра, технология Graftroot или вообще другие системы сценариев вроде Simplicity. Протокол Segwit предоставляет ясный путь к обновлению, и программное обеспечение, способное проверить достоверность версии Segwit X, будет проверять подобные транзакции на достоверность. А программное обеспечение, не способное проверить достоверность версии Segwit X, просто будет выполнять обработку вплоть до того места, где вступает в действие особое правило.

Сценарий p2sh-p2wpkh

Сценарий p2wpkh замечателен, но, к сожалению, это новый тип сценария, а прежние кошельки не способны пересылать биткойны по сценариям ScriptPubKey для p2wpkh. В p2wpkh применяется новый формат адресов Bech32, определенный в протоколе BIP0173, но в сценариях ScriptPubKey прежних кошельков неизвестно, как формировать такие адреса.

Авторы Segwit изобрели искусный способ обеспечить обратную совместимость Segwit, "заключив" p2wpkh в оболочку p2sh. Это так называемый "вложенный" протокол Segwit, поскольку сценарий ScriptPubKey вложен в сценарий RedeemScript для p2sh.

Адрес p2sh-p2wpkh является обычным адресом p2sh, но поле RedeemScript содержит последовательность команд `OP_0` *<20-байтовый хеш>* или сценарий ScriptPubKey для p2wpkh. Как и в p2wpkh, транзакции отсылаются старым узлам (рис. 13.9), как и новым узлам (рис. 13.10).

```
0100000001712e5b4e97ab549d50ca60a4f5968b2225215e9fab82dae4720078711406972f0000000
0171600148482O2fc47fb475289652fbd1912cc853ecb0096feffffff02323600000000000001976a9
14121ae7a2d55d2f0102ccc117cbcb70041b0e037f88ac1027000000000001976a914ec0be509516
51261765cfa71d7bd41c7b9245bb388ac075a0700
```

- 01000000 - версия
- 01 - # количество входов
- 712e...2f - хеш предыдущей транзакции
- 00000000 - индекс предыдущей транзакции
- 1716...96 - ScriptSig
- feffffff - последовательность
- 02 - # количество выходов
- 3236...00 - суммы на выходах
- 1976...ac - ScriptPubKey
- 075a0700 - время блокировки

Рис. 13.9. Оплата по хешу сценария и открытого ключа с раздельным заверением (p2sh-p2wpkh) до внедрения программного обеспечения по протоколу BIP0141

```
0100000000101712e5b4e97ab549d50ca60a4f5968b2225215e9fab82dae4720078711406972f000
0000017160014848202fc47fb475289652fbd1912cc853ecb0096feffffff02323600000000000019
76a914121ae7a2d55d2f0102ccc117cbcb70041b0e037f88ac1027000000000001976a914ec0be50
951651261765cfa71d7bd41c7b9245bb388ac02483045022100926 3c7de80c297d5b21aba846cf6f0
a970e1d339568167d1e4c1355c7711bc1602202c9312b8d32fd9c7acc54c46cab50eb7255ce3c0122
14c41fe1ad91bccb16a13012102ebdf6fc448431a2bd6380f912a0fa6ca291ca3340e79b6f0c1fdaf
f73cf54061075a0700
```

- 01000000 - версия
- 00 - маркер Segwit
- 01 - признак Segwit
- 01 - # количество входов
- 712e...2f - хеш предыдущей транзакции
- 00000000 - индекс предыдущей транзакции
- 1716...96 - ScriptSig
- feffffff - последовательность
- 02 - # количество выходов
- 3236...00 - суммы на выходах
- 1976...ac - ScriptPubKey
- 0248...61 - заверение
- 075a0700 - время блокировки

Рис. 13.10. Оплата по хешу сценария и открытого ключа с раздельным заверением (p2sh-p2wpkh) после внедрения программного обеспечения по протоколу BIP0141

Отличие рассматриваемого здесь вложенного сценария от p2wpkh состоит в том, что поле ScriptSig больше не является пустым. В нем содержится сценарий RedeemScript, что равнозначно сценарию ScriptPubkey в p2wpkh. А поскольку это часть сценария p2sh, поле ScriptPubkey оказывается таким же,

как и в любом другом сценарии p2wpkh. На рис. 13.11 показано, как выглядит объединенный сценарий.

Рис. 13.11. Поле ScriptPubKey в p2sh-p2wpkh оказывается таким же, как и обычное поле ScriptPubKey в p2sh

Вычисление объединенного сценария начинается так, как показано на рис. 13.12.

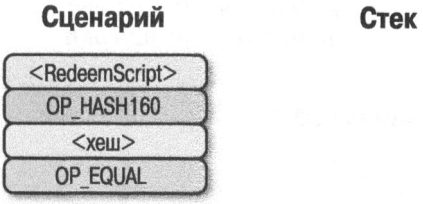

Рис. 13.12. Начальное состояние сценария p2sh-p2wpkh

Следует, однако, иметь в виду, что команды, которые должны обрабатываться, оказываются такими же, как и те, которые приводят в действие особое правило в p2sh. В частности, сценарий RedeemScript направляется в стек (рис. 13.13).

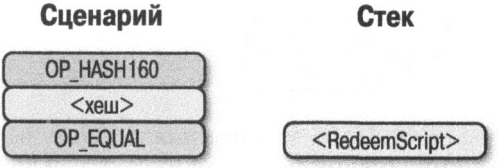

Рис. 13.13. Первая стадия выполнения сценария p2sh-p2wpkh

Операция `OP_HASH160` возвратит хеш по сценарию RedeemScript (рис. 13.14).

Полученный в итоге хеш направится в стек, а затем будет выполнена команда `OP_EQUAL` (рис. 13.15).

Если на данной стадии хеши окажутся одинаковыми, то в старых узлах (до внедрения протокола BIP0016) ввод транзакции будет просто отмечен как

достоверный, поскольку в этих узлах неизвестны правила проверки достоверности, применяемые в p2sh. А в новых узлах (после внедрения протокола BIP0016) распознается особая последовательность для сценария p2sh, и поэтому сценарий RedeemScript будет затем вычислен в виде команд Script. Сценарий RedeemScript состоит из последовательности команд `OP_0 <20-байтовый хеш>`, что равнозначно сценарию ScriptPubKey для p2wpkh. Таким образом, состояние сценария на данной стадии будет выглядеть так, как показано на рис. 13.16.

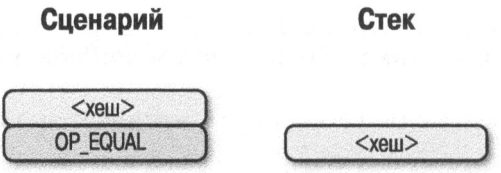

Рис. 13.14. Вторая стадия выполнения сценария p2sh-p2wpkh

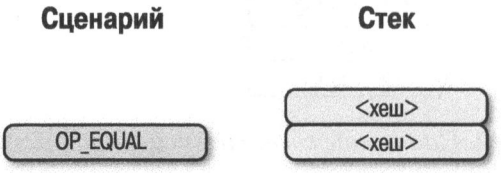

Рис. 13.15. Третья стадия выполнения сценария p2sh-p2wpkh

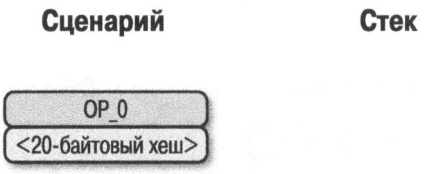

Рис. 13.16. Четвертая стадия выполнения сценария p2sh-p2wpkh

Это должно быть уже знакомо, поскольку напоминает состояние, с которого начинается выполнение сценария p2wpkh. После операции `OP_0` и 20-байтового хеша в стеке остается нуль, как показано на рис. 13.17.

На этой стадии старые узлы (до внедрения протокола Segwit) отметят данный ввод транзакции как достоверный, поскольку им неизвестны правила проверки достоверности, принятые в Segwit. А вот новые узлы (после

внедрения протокола Segwit) распознают особую последовательность для сценария p2wpkh. Подпись и открытый ключ из поля заверения наряду с 20-байтовым хешем будут введены в набор команд для сценария p2pkh (рис. 13.18).

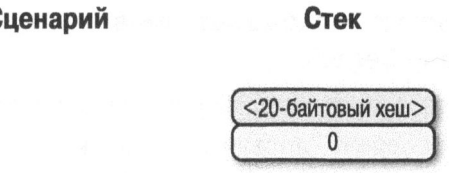

Рис. 13.17. Пятая стадия выполнения сценария p2sh-p2wpkh

Рис. 13.18. Шестая стадия выполнения сценария p2sh-p2wpkh

Остальная часть обработки рассматриваемого здесь сценария такая же, как и для p2pkh (см. главу 6). Если подпись и открытый ключ признаны действительными, то в стеке остается 1, как показано на рис. 13.19.

Рис. 13.19. Конечное состояние сценария p2sh-p2wpkh

Как видите, транзакции по вложенному сценарию p2sh-p2wpkh являются обратно совместимыми вплоть до внедрения протокола BIP0016. Более старые узлы (до внедрения протокола BIP0016) посчитают сценарий достоверным, если сценарии RedeemScript оказались одинаковыми. А менее старые узлы (после внедрения протокола BIP0016, но до внедрения протокола Segwit) посчитают сценарий достоверным по 20-байтовому хешу. Впрочем, и те, и другие узлы выполнят проверку достоверности не полностью и примут

транзакцию. И наконец, новые узлы (после внедрения протокола Segwit) выполнят проверку достоверности полностью, в том числе проверку подписи и открытого ключа.

Все ли могут расходовать выводы транзакций по протоколу Segwit

Противники протокола Segwit назвали выводы транзакций по протоколу Segwit "доступными всякому для расходования". Это было бы верно, если бы биткойн-сообщество отвергло Segwit. Иными словами, если бы материально заинтересованная часть биткойн-сообщества отказалась выполнять проверку достоверности по протоколу Segwit и активно отделилась от сети, принимая только те транзакции, которые были недостоверны по протоколу Segwit, то выводы таких транзакций были бы доступны для расходования всякому. Тем не менее протокол Segwit был внедрен по самым разным причинам экономического характера, а разделения сети не произошло, и немало биткойнов теперь заблокированы на вводах транзакций Segwit, которые проверены на достоверность по правилам "мягкой вилки" в большинстве экономически заинтересованных узлов. И теперь можно с уверенностью сказать, что противники протокола Segwit были не правы.

Программная реализация сценариев p2wpkh и p2sh-p2wpkh

Первое изменение мы собираемся сделать в классе `Tx`, где требуется пометить транзакцию как относящуюся к Segwit или другому протоколу:

```
class Tx:
  command = b'tx'

  def __init__(self, version, tx_ins, tx_outs,
    locktime, testnet=False, segwit=False):
    self.version = version
    self.tx_ins = tx_ins
    self.tx_outs = tx_outs
    self.locktime = locktime
    self.testnet = testnet
    self.segwit = segwit
```

```
    self._hash_prevouts = None
    self._hash_sequence = None
    self._hash_outputs = None
```

Затем внесем изменения в метод `parse()` в зависимости от получаемого результата сериализации, как показано ниже.

```
class Tx:
...
  @classmethod
  def parse(cls, s, testnet=False):
    s.read(4)    ❶
    if s.read(1) == b'\x00':    ❷
      parse_method = cls.parse_segwit
    else:
      parse_method = cls.parse_legacy
    s.seek(-5, 1)    ❸
    return parse_method(s, testnet=testnet)

  @classmethod
  def parse_legacy(cls, s, testnet=False):
    version = little_endian_to_int(s.read(4))    ❹
    num_inputs = read_varint(s)
    inputs = []
    for _ in range(num_inputs):
      inputs.append(TxIn.parse(s))
    num_outputs = read_varint(s)
    outputs = []
    for _ in range(num_outputs):
      outputs.append(TxOut.parse(s))
    locktime = little_endian_to_int(s.read(4))
    return cls(version, inputs, outputs, locktime,
               testnet=testnet, segwit=False)
```

❶ Проанализировать пятый байт, чтобы выяснить, относится ли проверяемая транзакция к протоколу Segwit. Первые четыре байта обозначают версию, а пятый байт — маркер Segwit.

❷ Если пятый байт равен нулю, это означает, что данная транзакция относится к протоколу Segwit (этот критерий используется здесь несмотря на то, что он ненадежен). В зависимости от того, относится ли транзакция к протоколу Segwit, здесь применяются разные синтаксические анализаторы.

❸ Вернуть поток в состояние, предшествующее проверке первых пяти байтов.

❹ Здесь прежний метод `parse` замещается новым методом `parse_legacy()`.

Ниже приведен синтаксический анализатор результата сериализации по протоколу Segwit.

```
class Tx:
...
  @classmethod
  def parse_segwit(cls, s, testnet=False):
    version = little_endian_to_int(s.read(4))
    marker = s.read(2)
    if marker != b'\x00\x01':     ❶
      raise RuntimeError('Not a segwit transaction {}'
                         .format(marker))
    num_inputs = read_varint(s)
    inputs = []
    for _ in range(num_inputs):
      inputs.append(TxIn.parse(s))
    num_outputs = read_varint(s)
    outputs = []
    for _ in range(num_outputs):
      outputs.append(TxOut.parse(s))
    for tx_in in inputs:     ❷
      num_items = read_varint(s)
      items = []
      for _ in range(num_items):
        item_len = read_varint(s)
        if item_len == 0:
          items.append(0)
        else:
          items.append(s.read(item_len))
      tx_in.witness = items
    locktime = little_endian_to_int(s.read(4))
    return cls(version, inputs, outputs, locktime,
               testnet=testnet, segwit=True)
```

❶ Здесь имеются два новых поля, и в одном из них содержится маркер Segwit.

❷ А во втором новом поле содержится заверение, состоящее из отдельных элементов для каждого ввода транзакции.

А теперь можно запрограммировать соответствующие изменения в методах сериализации:

```
class Tx:
...
  def serialize(self):
     if self.segwit:
       return self.serialize_segwit()
```

```
    else:
      return self.serialize_legacy()

  def serialize_legacy(self):     ❶
    result = int_to_little_endian(self.version, 4)
    result += encode_varint(len(self.tx_ins))
    for tx_in in self.tx_ins:
      result += tx_in.serialize()
      result += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
      result += tx_out.serialize()
    result += int_to_little_endian(self.locktime, 4)
    return result

  def serialize_segwit(self):
    result = int_to_little_endian(self.version, 4)
    result += b'\x00\x01'     ❷
    result += encode_varint(len(self.tx_ins))
    for tx_in in self.tx_ins:
      result += tx_in.serialize()
    result += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
      result += tx_out.serialize()
      for tx_in in self.tx_ins:     ❸
        result += int_to_little_endian(len(tx_in.witness), 1)
        for item in tx_in.witness:
          if type(item) == int:
            result += int_to_little_endian(item, 1)
        else:
          result += encode_varint(len(item)) + item
    result += int_to_little_endian(self.locktime, 4)
    return result
```

❶ Здесь определяется метод `serialize_legacy()`, который раньше назывался `erialize()`.

❷ Сериализация по протоколу Segwit дополняется здесь соответствующими маркерами.

❸ Под конец сериализируется заверение.

Необходимо внести изменения и в метод `hash()`, как показано ниже, чтобы воспользоваться прежним механизмом сериализации — даже для транзакций Segwit, поскольку это обеспечит устойчивость идентификатора транзакции.

```
class Tx:
...
  def hash(self):
    '''Двоичный хеш-код прежней сериализации'''
    return hash256(self.serialize_legacy())[::-1]
```

В приведенном ниже методе `verify_input()` требуется другой хеш подписи z для транзакций Segwit. Порядок вычисления хеша подписи для транзакции Segwit определяется в протоколе BIP0143. Помимо него, механизму вычисления сценариев передается поле заверения.

```
class Tx:
...
  def verify_input(self, input_index):
    tx_in = self.tx_ins[input_index]
    script_pubkey = tx_in.script_pubkey(testnet=self.testnet)
    if script_pubkey.is_p2sh_script_pubkey():
      command = tx_in.script_sig.commands[-1]
      raw_redeem = int_to_little_endian(len(command), 1) + command
      redeem_script = Script.parse(BytesIO(raw_redeem))
      if redeem_script.is_p2wpkh_script_pubkey():     ❶
        z = self.sig_hash_bip143(input_index, redeem_script)     ❷
        witness = tx_in.witness
      else:
        z = self.sig_hash(input_index, redeem_script)
        witness = None
    else:
      if script_pubkey.is_p2wpkh_script_pubkey():     ❸
        z = self.sig_hash_bip143(input_index)
        witness = tx_in.witness
      else:
        z = self.sig_hash(input_index)
        witness = None
    combined_script = tx_in.script_sig
                    + tx_in.script_pubkey(self.testnet)
    return combined_script.evaluate(z, witness)     ❹
```

❶ Здесь проверяется вариант вложенного сценария p2sh-p2wpkh.

❷ Код для генерирования хеша подписи по протоколу BIP0143 находится в исходном файле `tx.py` из примеров кода к этой главе.

❸ Здесь проверяется вариант сценария p2wpkh.

❹ Заверение передается механизму вычисления сценариев, и поэтому по сценарию p2wpkh можно построить подходящие команды.

Необходимо также определить сценарий в исходном файле `script.py`, как показано ниже.

```
def p2wpkh_script(h160):
  '''Принимает хеш-код hash160 и возвращает
     сценарий ScriptPubKey для p2wpkh'''
  return Script([0x00, h160])     ❶
...
def is_p2wpkh_script_pubkey(self):     ❷
  return len(self.cmds) == 2 and self.cmds[0] == 0x00 \
                             and type(self.cmds[1]) == bytes \
                             and len(self.cmds[1]) == 20
```

❶ Это последовательность команд `OP_0 <20-байтовый хеш>`.

❷ Здесь проверяется, относится ли текущий сценарий к типу ScriptPubKey для p2wpkh.

И наконец, необходимо реализовать особое правило в методе `evaluate()` следующим образом:

```
class Script:
...
  def evaluate(self, z, witness):
  ...
    while len(commands) > 0:
    ...
      else:
        stack.append(command)
        ...
        if len(stack) == 2 and stack[0] == b'' \
                           and len(stack[1]) == 20:     ❶
          h160 = stack.pop()
          stack.pop()
          cmds.extend(witness)
          cmds.extend(p2pkh_script(h160).cmds)
```

❶ Именно здесь выполняется нулевая версия сценария заверения для p2wpkh. Объединенный сценарий p2pkh сначала создается из 20-байтового хеша, подписи и открытого ключа, а затем вычисляется.

Оплата по хешу сценария с раздельным заверением (p2wsh)

Несмотря на то что сценарий p2wpkh служит основным вариантом оплаты с заверением, все же требуется более гибкий способ выполнения усложненных сценариев (например, с мультиподписями). Именно здесь и приходит на помощь способ оплаты по хешу сценария с раздельным заверением (p2wsh).

Сценарий p2wsh подобен сценарию p2sh, но все данные переносятся в нем из поля ScriptSig в поле заверения.

Как и в p2wpkh, данные отсылаются старому программному обеспечению (до внедрения протокола BIP0141; рис. 13.20), а не новому программному обеспечению (после внедрения протокола BIP0141; рис. 13.21).

```
01000000 01 593a2db37b841b2a46f4e9bb63fe9c1012da3ab7fe30b9f9c974242778b5af898000000
0000 ffffffff 01 806fb30700000001976a914bbef244bcad13cffb68b5cef3017c7423675552288a
c00000000

        - 01000000 - версия
        - 01 - # количество входов
        - 593a...98 - хеш предыдущей транзакции
        - 00000000 - индекс предыдущей транзакции
        - 00 - ScriptSig
        - ffffffff - последовательность
        - 01 - # количество выходов
        - 806f...00 - суммы на выходах
        - 1976...ac - ScriptPubKey
        - 00000000 - время блокировки
```

Рис. 13.20. Оплата по хешу сценария с раздельным заверением, распознаваемая старым программным обеспечением (до внедрения протокола BIP0141)

```
01000000 00 01 01 593a2db37b841b2a46f4e9bb63fe9c1012da3ab7fe30b9f9c974242778b5af89800
00000000 ffffffff 01 806fb30700000001976a914bbef244bcad13cffb68b5cef3017c7423675552
288ac 04004730440220 3cdcaf02a44e37e409646e8a506724e9e1394b890cb52429ea65bac4cc2403
f1022024b934297bcd0c21f22cee0e48751c8b184cc3a0d704cae2684e14858550af7d01483045022
100feb4e1530c13e72226dc912dcd257df90d81ae22dbddb5a3c2f6d86f81d47c8e022069889ddb76
388fa7948aaa018b2480ac36132009bb9cfade82b651e88b4b137a01695221026ccfb8061f235cc11
0697c0bfb3afb99d82c886672f6b9b5393b25a434c0cbf32103befa190c0c22e2f53720b1be9476dc
f11917da4665c44c9c71c3a2d28a933c352102be46dc245f58085743b1cc37c82f0d63a960efa43b5
336534275fc469b49f4ac53ae 00000000

        - 01000000 - версия
        - 00 - маркер Segwit
        - 01 - признак Segwit
        - 01 - # количество входов
        - 593a...98 - хеш предыдущей транзакции
        - 00000000 - индекс предыдущей транзакции
        - 00 - ScriptSig
        - ffffffff - последовательность
        - 01 - # количество выходов
        - 806f...00 - суммы на выходах
        - 1976...ac - ScriptPubKey
        - 0400...ae - заверение
        - 00000000 - время блокировки
```

Рис. 13.21. Оплата по хешу сценария с раздельным заверением, распознаваемая новым программным обеспечением (после внедрения протокола BIP0141)

Сценарий ScriptPubKey для p2wsh состоит из последовательности команд `OP_0 <32-байтовый хеш>`. Эта последовательность команд приводит в действие другое особое правило, а поле ScriptSig, как и в сценарии p2wpkh, остается пустым. Если выводы транзакций отсылаются по сценарию p2wsh, то объединенный сценарий выглядит так, как показано на рис. 13.22.

Рис. 13.22. Поля ScriptPubKey и ScriptSig для оплаты по хешу сценария с раздельным заверением (p2wpkh)

Обработка данного сценария начинается таким же образом, как и сценария p2wpkh (рис. 13.23 и 13.24).

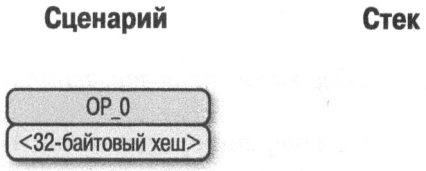

Рис. 13.23. Начальное состояние сценария p2wpkh

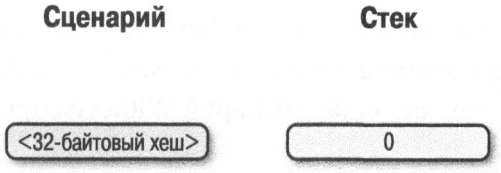

Рис. 13.24. Первая стадия выполнения сценария p2wpkh

32-байтовый хеш является элементом и поэтому размещается в стеке (рис. 13.25).

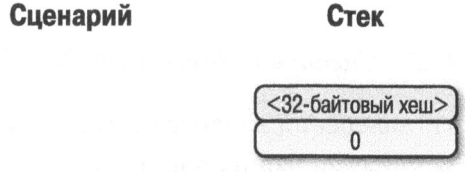

Рис. 13.25. Вторая стадия выполнения сценария p2wpkh

Как и в p2wpkh, старые узлы остановятся на этом, поскольку больше не останется команд Script для обработки. А новые узлы распознают особую последовательность команд и выполнят дополнительную проверку на достоверность, проанализировав содержимое поля заверения. В данном случае поле заверения для p2wsh будет содержать мультиподпись типа "2 из 3" (рис. 13.26).

```
040047304402203cdcaf02a44e37e409646e8a506724e9e1394b890cb52429ea65bac4cc2403f1022
024b934297bcd0c21f22cee0e48751c8b184cc3a0d704cae2684e14858550af7d01483045022100fe
b4e1530c13e72226dc912dcd257df90d81ae22dbddb5a3c2f6d86f81d47c8e022069889ddb76388fa
7948aaa018b2480ac36132009bb9cfade82b651e88b4b137a01695221026ccfb8061f235cc110697c
0bfb3afb99d82c886672f6b9b5393b25a434c0cbf32103befa190c0c22e2f53720b1be9476dcf1191
7da4665c44c9c71c3a2d28a933c352102be46dc245f58085743b1cc37c82f0d63a960efa43b533653
4275fc469b49f4ac53ae

- 04 - Количество элементов заверения
- 00 - OP_0
- 47 - Длина <подписи x>
- 3044...01 - <подпись x>
- 69 - Длина поля WitnessScript
- 5221...ae - <WitnessScript>
```

Рис. 13.26. Содержимое поля заверения для p2wsh

Последний элемент в поле заверения называется *WitnessScript*, и он должен быть 32-байтовым хеш-кодом sha256, получаемым из поля ScriptPubKey. Следует, однако, иметь в виду, что этот хеш-код получается по алгоритму sha256, а не hash256. Как только элемент WitnessScript пройдет проверку достоверности на то же самое значение хеша, он будет интерпретирован как последовательность команд сценария, вводимую в набор выполняемых команд. На рис. 13.27 показано, как выглядит сценарий WitnessScript.

```
5221026ccfb8061f235cc110697c0bfb3afb99d82c886672f6b9b5393b25a434c0cbf32103befa190
c0c22e2f53720b1be9476dcf11917da4665c44c9c71c3a2d28a933c352102be46dc245f58085743b1
cc37c82f0d63a960efa43b5336534275fc469b49f4ac53ae

- 52 - OP_2
- 21 - Длина <открытого ключа x>
- 026c...f3 - <открытый ключ x>
- 53 - OP_3
- ae - OP_CHECKMULTISIG
```

Рис. 13.27. Сценарий WitnessScript для p2wsh

Остальная часть поля заверения вводится в стек сверху, чтобы в итоге получился набор команд, приведенный на рис. 13.28.

Рис. 13.28. Третья стадия выполнения сценария p2wpkh

Как видите, это мультиподпись типа "2 из 3", аналогичная той, которая рассматривалась в главе 8 (рис. 13.29).

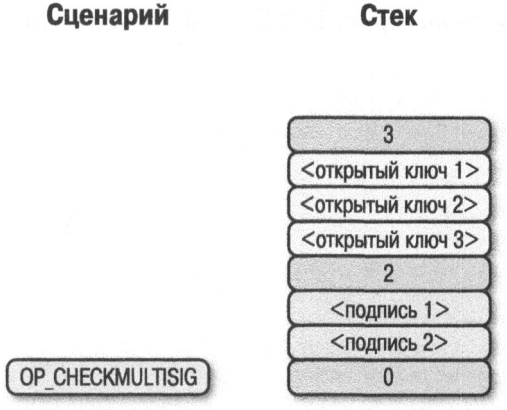

Рис. 13.29. Четвертая стадия выполнения сценария p2wpkh

Если подписи действительны, то данный сценарий завершается так, как показано на рис. 13.30.

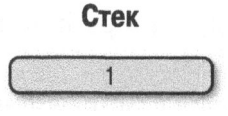

Рис. 13.30. Пятая стадия выполнения сценария p2wpkh

Сценарий WitnessScript очень похож на сценарий RedeemScript в том отношении, что обращение к хеш-коду sha256 сериализации осуществляется в поле ScriptPubKey, хотя он выявляется лишь в том случае, если отсылается вывод транзакции. И как только хеш-код sha256 из сценария WitnessScript будет выявлен таким же, как и 32-байтовый хеш, WitnessScript интерпретируется как последовательность команд сценария, вводимая в набор команд. А остальная часть поля заверения добавляется затем к набору команд, образуя тем самым окончательный набор выполняемых команд. Сценарий p2wsh имеет особое значение, поскольку для создания двунаправленных платежных каналов по протоколу Lightning Network требуются неподатливые транзакции с мультиподписями.

Сценарий p2sh-p2wsh

Подобно вложенному сценарию p2sh-p2wpkh, сценарий p2sh-p2wsh обеспечивает обратную совместимость p2wsh. И в этом случае старым узлам (рис. 13.31) и новым узлам (рис. 13.32) отсылаются разные транзакции.

```
01000000017082 56c5896fb3f00ef37601f8e30c5b460dbcd1fca1cd7199f9b56fc4ecd5400
0000023220020615ae01ed1bc1ffaad54da31d7805d0bb55b52dfd3941114330368c1bbf69
b4cffffffff01603edb0300000000160014bbef244bcad13cffb68b5cef3017c74236755522
00000000

        - 01000000 - версия
        - 01 - # количество входов
        - 7082...54 - хеш предыдущей транзакции
        - 00000000 - индекс предыдущей транзакции
        - 2322...4c - ScriptSig
        - ffffffff - последовательность
        - 01 - # количество выходов
        - 603e...00 - суммы на выходах
        - 1600...22 - ScriptPubKey
        -
```

Рис. 13.31. Оплата по хешу сценария и с раздельным заверением (p2sh-p2wsh) для программного обеспечения до внедрения протокола BIP0141

Как в сценарии p2sh-p2wpkh, поле ScriptPubKey ничем не отличается от любого другого адреса в p2sh, а поле ScriptSig содержит лишь сценарий RedeemScript (рис. 13.33).

Вычисление сценария p2sh-p2wsh начинается точно так же, как и сценария p2sh-p2wpkh (рис. 13.34).

```
0100000000010170825 6c5896fb3f00ef37601f8e30c5b460dbcd1fca1cd7199f9b56fc4ecd540000
00002322002061 5ae01ed1bc1ffaad54da31d7805d0bb55b52dfd3941114330368c1bbf69b4cfffff
fff01603edb0300000000160014bbef244bcad13cffb68b5cef3017c74236755522040047304402 0
010d2854b86b90b7c33661ca25f9d9f15c24b88c5c4992630f77ff004b998fb802204106fc3ec8481
fa98e07b7e78809ac91b6ccaf60bf4d3f729c5a75899bb664a501473044022046d66321c6766abcb1
366a793f9bfd0e11e0b080354f18188588961ea76c5ad002207262381a0661d66f5c39825202524c4
5f29d500c6476176cd910b1691176858701695221026ccfb8061f235cc110697c0bfb3afb99d82c88
6672f6b9b5393b25a434c0cbf32103befa190c0c22e2f53720b1be9476dcf11917da4665c44c9c71c
3a2d28a933c352102be46dc245f58085743b1cc37c82f0d63a960efa43b5336534275fc469b49f4ac
53ae00000000
```

- 01000000 - версия
- 00 - маркер Segwit
- 01 - признак Segwit
- 01 - # количество входов
- 7082...54 - хеш предыдущей транзакции
- 00000000 - индекс предыдущей транзакции
- 2322...4c - ScriptSig
- ffffffff - последовательность
- 01 - # количество выходов
- 603e...00 - суммы на выходах
- 1600...22 - ScriptPubKey
- 0400...ae - заверение
- 00000000 - время блокировки

Рис. 13.32. Оплата по хешу сценария и с раздельным заверением (p2sh-p2wsh) для программного обеспечения после внедрения протокола BIP0141

Рис. 13.33. Поля ScriptPubKey и ScriptSig для оплаты по хешу сценария и с раздельным заверением (p2sh-p2wsh)

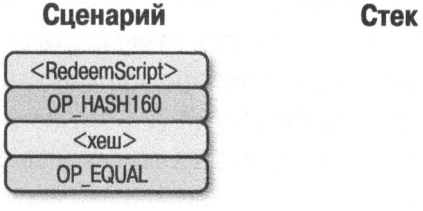

Рис. 13.34. Начальное состояние сценария p2sh-p2wsh

Затем сценарий RedeemScript размещается в стеке (рис. 13.35).

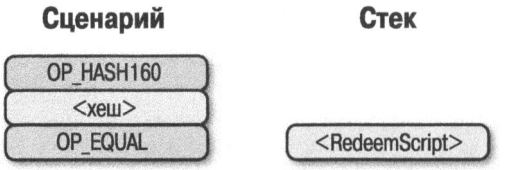

Рис. 13.35. Первая стадия выполнения сценария p2sh-p2wsh

Операция `OP_HASH160` возвратит хеш из сценария RedeemScript (рис. 13.36).

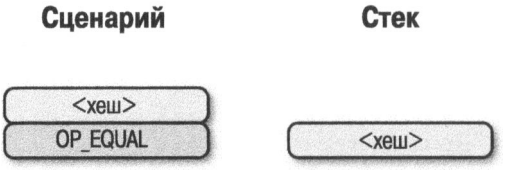

Рис. 13.36. Вторая стадия выполнения сценария p2sh-p2wsh

Далее хеш размещается в стеке, после чего выполняется операция `OP_EQUAL` (13.37).

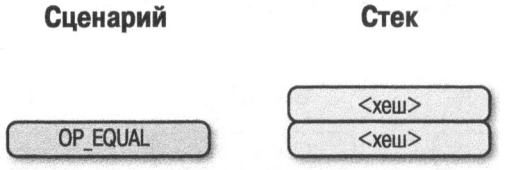

Рис. 13.37. Третья стадия выполнения сценария p2sh-p2wsh

Как в сценарии p2sh-p2wpkh, старые узлы (до внедрения протокола BIP0016) отметят ввод транзакции как достоверный, если сравниваемые хеши равны, поскольку им ничего неизвестно о правилах проверки на достоверность в p2sh. А новые узлы (после внедрения протокола BIP0016) распознают особую последовательность команд для выполнения сценария p2sh, и поэтому сценарий RedeemScript будет интерпретирован как новые команды сценария. В частности, сценарий RedeemScript состоит из такой же самой последовательности команд `OP_0` *32-байтовый хеш*, как и сценарий ScriptPubKey для p2wsh (рис. 13.38).

В итоге состояние сценария окажется таким, как показано на рис. 13.39.

И это, конечно, такое же самое начальное состояние, как и для сценария p2wsh (13.40).

220020615ae01ed1bc1ffaad54da31d7805d0bb55b52dfd3941114330368c1bbf69b4c

- 22 - Длина поля <RedeemScript>
- 0020...4c - <RedeemScript>

Рис. 13.38. Сценарий RedeemScript для p2sh-p2wpkh

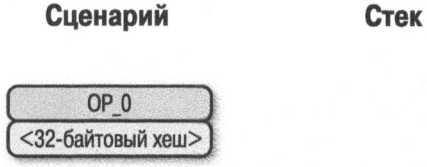

Рис. 13.39. Четвертая стадия выполнения сценария p2sh-p2wsh

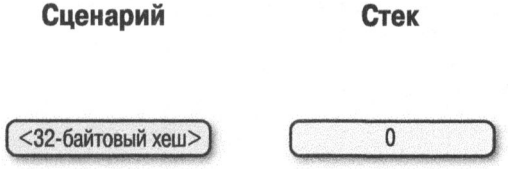

Рис. 13.40. Пятая стадия выполнения сценария p2sh-p2wsh

32-байтовый хеш является элементом и поэтому размещается в стеке (рис. 13.41).

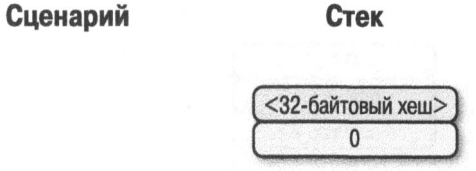

Рис. 13.41. Шестая стадия выполнения сценария p2sh-p2wsh

На этой стадии старые узлы (до внедрения протокола Segwit) отметят данный ввод транзакции как достоверный, поскольку им неизвестны правила проверки на достоверность, принятые в Segwit. А новые узлы (после внедрения протокола Segwit) распознают особую последовательность команд для выполнения сценария p2wsh. Поле заверения (рис. 13.42) содержит сценарий WitnessScript (13.43). Хеш-код sha256 из сценария WitnessScript сравнивается с 32-байтовым хешем, и если они равны, то WitnessScript интерпретируется как команды выполнения сценария, вводимые в набор команд (рис. 13.44).

```
040047304402200l0d2854b86b90b7c33661ca25f9d9f15c24b88c5c4992630f77ff004
b998fb802204106fc3ec8481fa98e07b7e78809ac91b6ccaf60bf4d3f729c5a75899bb6
64a5014730440220460d66321c6766abcb1366a793f9bfd0e11e0b080354f18188588961
ea76c5ad002207262381a0661d66f5c39825202524c45f29d500c6476176cd910b16911
76858701695221026ccfb8061f235cc110697c0bfb3afb99d82c886672f6b9b5393b25a
434c0cbf32103befa190c0c22e2f53720b1be9476dcf11917da4665c44c9c71c3a2d28a
933c352102be46dc245f58085743b1cc37c82f0d63a960efa43b5336534275fc469b49f
4ac53ae
```

- 04 - Количество элементов заверения
- 00 - OP_0
- 47 - Длина <подписи x>
- 3044...01 - <подпись x>
- 69 - Длина поля WitnessScript
- 5221...ae - <WitnessScript>

Рис. 13.42. Поле заверения из сценария p2sh-p2wsh

```
5221026ccfb8061f235cc110697c0bfb3afb99d82c886672f6b9b5393b25a434c0cbf32103befa190
c0c22e2f53720b1be9476dcf11917da4665c44c9c71c3a2d228a933c352102be46dc245f58085743b
1cc37c82f0d63a960efa43b5336534275fc469b49f4as53ae
```

- 52 - OP_2
- 21 - Длина <открытого ключа x>
- 026c...f3 - <открытый ключ x>
- 53 - OP_3
- ae - OP_CHECKMULTISIG

Рис. 13.43. Сценарий WitnessScript для p2sh-p2wsh

Рис. 13.44. Седьмая стадия выполнения сценария p2sh-p2wsh

Как видите, это мультиподпись типа "2 из 3", аналогичная рассматриваемой в главе 8. Если подписи действительны, то данный сценарий завершается так, как показано на рис. 13.45.

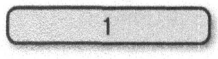

Рис. 13.45. Конечное состояние сценария p2sh-p2wsh

Подобным образом обеспечивается обратная совместимость сценария p2wsh. А прежние кошельки могут пересылать биткойны для оплаты типа p2sh по сценариям ScriptPubKey.

Программная реализация сценариев p2wsh и p2sh-p2wsh

Синтаксический анализ и сериализация реализуются в данном случае таким же образом, как и прежде. Основные изменения придется внести в метод `verify_input()` из исходного файла `tx.py`, а также в метод `evaluate()` из исходного файла `script.py`, как показано ниже.

```
class Tx:
...
  def verify_input(self, input_index):
    tx_in = self.tx_ins[input_index]
    script_pubkey = tx_in.script_pubkey(testnet=self.testnet)
    if script_pubkey.is_p2sh_script_pubkey():
      command = tx_in.script_sig.commands[-1]
      raw_redeem = int_to_little_endian(len(command), 1) \
                   + command
      redeem_script = Script.parse(BytesIO(raw_redeem))
      if redeem_script.is_p2wpkh_script_pubkey():
        z = self.sig_hash_bip143(input_index, redeem_script)
        witness = tx_in.witness
      elif redeem_script.is_p2wsh_script_pubkey():     ❶
        command = tx_in.witness[-1]
        raw_witness = encode_varint(len(command)) + command
        witness_script = Script.parse(BytesIO(raw_witness))
        z = self.sig_hash_bip143(input_index,
        witness_script=witness_script)
        witness = tx_in.witness
      else:
        z = self.sig_hash(input_index, redeem_script)
        witness = None
    else:
      if script_pubkey.is_p2wpkh_script_pubkey():
        z = self.sig_hash_bip143(input_index)
```

```
      witness = tx_in.witness
    elif script_pubkey.is_p2wsh_script_pubkey():     ❷
      command = tx_in.witness[-1]
      raw_witness = encode_varint(len(command)) + command
      witness_script = Script.parse(BytesIO(raw_witness))
      z = self.sig_hash_bip143(input_index,
                               witness_script=witness_script)
      witness = tx_in.witness
    else:
      z = self.sig_hash(input_index)
      witness = None
  combined_script = tx_in.script_sig \
                  + tx_in.script_pubkey(self.testnet)
  return combined_script.evaluate(z, witness)
```

❶ Здесь обрабатывается сценарий p2sh-p2wsh.

❷ Здесь обрабатывается сценарий p2wsh.

Затем следует запрограммировать способ выявления сценария p2wsh в исходном файле `script.py`:

```
def p2wsh_script(h256):
  '''Принимает хеш-код hash160 и возвращает
     сценарий ScriptPubKey для p2wsh'''
  return Script([0x00, h256])     ❶
...
class Script:
...
  def is_p2wsh_script_pubkey(self):
    return len(self.cmds) == 2 and self.cmds[0] == 0x00 \
                               and type(self.cmds[1]) == bytes \
                               and len(self.cmds[1]) == 32
```

❶ Как и предполагалось, последовательность команд `OP_0 <32-байтовый хеш>`.

И наконец, ниже реализуется особое правило для p2wsh.

```
class Script:
...
  def evaluate(self, z, witness):
  ...
    while len(commands) > 0:
    ...
    else:
      stack.append(command)
      ...
```

```
    if len(stack) == 2 and stack[0] == b'' \
                       and len(stack[1]) == 32:
      s256 = stack.pop()     ❶
      stack.pop()     ❷
      cmds.extend(witness[:-1])     ❸
      witness_script = witness[-1]     ❹
      if s256 != sha256(witness_script):     ❺
        print('bad sha256 {} vs {}'.format(
               s256.hex(), sha256(witness_script).hex()))
        return False
      stream = BytesIO(encode_varint(len(witness_script))
                       + witness_script)
      witness_script_cmds = Script.parse(stream).cmds     ❻
      cmds.extend(witness_script_cmds)
```

❶ Элемент на вершине стека является хеш-кодом sha256 из сценария WitnessScript.

❷ Второй элемент в стеке обозначает нулевую версию заверения.

❸ Все, кроме сценария WitnessScript, вводится в набор команд.

❹ Сценарий WitnessScript является последним элементом в поле заверения.

❺ Сценарий WitnessScript следует хешировать по алгоритму sha256, как обозначено в стеке.

❻ Проанализировать синтаксически сценарий WitnessScript и ввести его в набор команд.

Прочие усовершенствования

В протоколе Segwit решается также задача квадратичного хеширования благодаря применению другого алгоритма вычисления хеша подписи. Немало вычислений хеша подписи (z) можно использовать повторно вместо того, чтобы требовать новый хеш-код hash256 для каждого ввода транзакции. Порядок, в котором вычисляется хеш подписи, подробно описан в протоколе BIP0143 и реализован в исходном файле `code-ch13/tx.py`.

Еще одно усовершенствование состоит в том, что по принятым правилам открытые ключи в несжатом формате SEC теперь запрещены. Ради экономии места в протоколе применяются открытые ключи только в сжатом формате SEC.

Заключение

В этой главе были подробно описаны особенности протокола Segwit, чтобы стали понятны его функциональные возможности. В главе 14 рассматриваются следующие шаги, которые вам предстоит предпринять на пути к разработке биткойна.

Дополнительные вопросы и следующие шаги

Если, прорабатывая материал данной книги, вы добрались до этой главы, можете себя поздравить! Это означает, что вы узнали немало о внутреннем механизме действия биткойна, и можно надеяться, что это вдохновило вас узнать об этой технологии еще больше. Данная тема затронута в этой книге лишь вскользь. А в этой главе будет вкратце рассмотрен ряд других вопросов, которые могут вас заинтересовать, чтобы изучить их более основательно. Кроме того, здесь поясняется, как начать карьеру разработчика биткойна и как присоединиться к сообществу разработчиков, активно действующих в данной области.

Темы для дальнейшего изучения

Ниже вкратце рассматриваются дополнительные вопросы и обсуждаются темы для дальнейшего изучения биткойна.

Криптовалютные кошельки

Создание криптовалютного кошелька — дело непростое, потому что защитить секретные ключи нелегко. Кроме того, имеется целый ряд стандартов, которые могут оказать помощь в создании криптовалютных кошельков.

Иерархически детерминированные кошельки

Из соображений конфиденциальности пользоваться адресами повторно не рекомендуется (см. главу 7). Это означает необходимость в создании целого ряда адресов. Но, к сожалению, хранение разных секретных ключей по каждому адресу в отдельности может стать непростой задачей обеспечения безопасности и резервного копирования. При этом возникают следующие

вопросы: как выполнить резервное копирование всех секретных ключей, как сгенерировать массу секретных ключей и выполнить их резервное копирование как впервые, так и повторно, какую систему выбрать, чтобы обеспечить актуальность резервных копий?

Для решения данной задачи были первоначально реализованы *детерминированные* кошельки, в том числе криптовалютный кошелек Armory. Суть детерминированного криптовалютного кошелька состоит в том, чтобы сгенерировать одно начальное случайное значение и создать по нему немало разных адресов. Детерминированные кошельки типа Armory были замечательны во многих отношениях, кроме необходимости каким-то образом группировать адреса. Поэтому был принят стандарт BIP0032 на *иерархически детерминированные* кошельки со многими уровнями и ключами, причем каждый со своим особым путем вывода. В стандарте BIP0032 были определены соответствующие спецификации и тестовые наборы, поэтому, реализовав собственный детерминированный кошелек в сети testnet, можно очень многому научиться.

Кроме того, в протоколе BIP0044 определяется, что именно означает каждый уровень иерархии по стандарту BIP0032, а также нормы передовой практики для применения единственного иерархически детерминированного начального случайного значения для хранения монет в самых разных криптовалютах. Реализация протокола BIP0044 позволяет также разобраться в инфраструктуре иерархически детерминированного кошелька. И хотя во многих кошельках (Trezor, Coinomi и т.д.) реализованы оба стандарта, BIP0032 и BIP0044, последний вообще игнорируется в некоторых кошельках, а вместо этого используется собственная иерархия по стандарту BIP0032 (примером тому служат криптовалютные кошельки Electrum и Edge).

Мнемонические начальные случайные значения

Записывать и перезаписывать 256-разрядные начальные случайные значения очень неудобно и чревато ошибками. В качестве вывода из этого положения в стандарте BIP0039 описывается способ кодирования начального случайного значения в виде целого ряда слов на английском языке. Таких слов может быть 2048 (т.е. 2^{11}), а это означает, что каждое слово кодирует 11 битов начального случайного значения. В стандарте BIP0039 точно определяется, каким образом зарезервированная мнемоника переводится в начальное случайное значение по стандарту BIP0032, а вместе со стандартами BIP0032 и BIP0044 — каким образом реализуются резервное копирование и восстановление большинства кошельков. Поэтому написание кода для

реализации криптовалютного кошелька, действующего в сети testnet по стандарту BIP0039, позволит вам войти во вкус разработки биткойна.

Платежные каналы и протокол Lightning Network

Платежные каналы являются атомарными единицами сети Lightning Network, и поэтому на следующем этапе очень полезно изучить принцип их действия. Реализовать платежные каналы можно самыми разными способами, но в стандарте BOLT описывается спецификация, применяемая в узлах сети Lightning Network. На момент написания данной книги эти спецификации находятся на стадии разработки и доступны по адресу `https://github.com/lightningnetwork/lightning-rfc/`.

Участие в разработке биткойна

Большая часть этики биткойна относится к активному участию в сообществе разработчиков. И основной способ сделать это — принять участие в проектах с открытым исходным кодом. Перечень таких проектов довольно обширный, и ниже в качестве примера перечислены некоторые из них.

- *Bitcoin Core* (`https://github.com/bitcoin/bitcoin`)

  Эталонный клиент
- *Libbitcoin* (`https://github.com/libbitcoin/libbitcoin-system`)

  Альтернативная реализация биткойна на C++
- *btcd* (`https://github.com/btcsuite/btcd`)

  Реализация биткойна на языке Golang
- *Bcoin* (`https://github.com/bcoin-org/bcoin`)

  Реализация биткойна на языке JavaScript с поддержкой со стороны вебслужбы purse.io
- *pycoin* (`https://github.com/richardkiss/pycoin`)

  Библиотека Python для биткойна
- *BitcoinJ* (`https://github.com/bitcoinj/bitcoinj`)

  Библиотека Java для биткойна
- *BitcoinJS* (`https://github.com/bitcoinjs/bitcoinjs-lib`)

  Библиотека JavaScript для биткойна

- *BTCPay* (`https://github.com/btcpayserver/btcpayserver`)

  Механизм обработки платежей в биткойнах, написанный на C#

Участие в подобных проектах может принести пользу по многим причинам, включая перспективные возможности трудоустройства, обучения, черпания замыслов по поводу открытия собственного дела и т.д.

Другие предлагаемые проекты

Если вы все еще раздумываете, в каком именно проекте вам выгоднее всего принять участие, обратите внимание на ряд других проектов, кратко описываемых в последующих разделах.

Криптовалютный кошелек в сети testnet

Важность обеспечения безопасности биткойна трудно переоценить. Написание кода, реализующего криптовалютный кошелек даже в сети testnet, поможет вам лучше понять различные соображения, которые следует принимать во внимание, создавая такой кошелек. Пользовательский интерфейс, резервные копии, адресные книги и предыстории транзакций являются лишь некоторыми вопросами, которые вам придется решать, создавая криптовалютный кошелек. А поскольку криптовалютный кошелек является самым распространенным приложением биткойна, создавая его, вы узнаете немало о потребностях пользователей.

Обозреватель блоков

Более честолюбивый проект подразумевает написание кода, реализующего обозреватель блоков. Ключом к созданию своего обозревателя блоков служит сохранение данных из блокчейна легко доступным способом. Здесь может пригодиться применение традиционной базы данных вроде Postgres или MySQL. А поскольку в Bitcoin Core отсутствуют индексы адресов, внедрив их, вы сможете искать неизрасходованные выводы транзакции (UTXO) и прошлые транзакции по адресу, что желательно для большинства пользователей.

Интернет-магазин

Интернет-магазин с торговыми операциями в биткойнах является еще одним проектом, работа над которым поможет вам в изучении данной технологии. Это особенно уместно для разработчиков веб-приложений, поскольку

им, как правило, известно, как создавать веб-приложения. Веб-приложение с серверной частью для поддержки биткойна может стать эффективным средством, позволяющим избегать сторонних зависимостей при оплате. Еще раз советуем начать с testnet и использовать криптографически защищенные библиотеки с безопасным туннельным соединением для выполнения платежей.

Служебная библиотека

Создание служебной библиотеки, подобной той, которая была построена в данной книге, служит еще одним отличным способом узнать больше о биткойне. В частности, написание кода для сериализации хеша подписи в протоколе Segwit по стандарту BIP0143 может оказать помощь в овладении программированием протоколов. А перенос кода из примеров, приведенных в данной книге, на другой язык программирования может также стать отличным подспорьем для изучения биткойна.

Трудоустройство

Если вы проявляете заинтересованность в более глубоком освоении данной области, то перед вами открываются обширные возможности стать профессиональным разработчиком. А подтверждением того, что вы кое-что знаете о биткойне, может стать портфель проектов, выполненных вами самостоятельно. Участие в действующих проектах с открытым кодом или выполнение собственного проекта поможет вам обратить на себя внимание работодателей. Кроме того, программирование на уровне прикладного интерфейса API, разработанного любой отдельной компанией, — отличный способ получить приглашение на собеседование по поводу трудоустройства!

Как правило, найти работу по месту жительства намного проще, поскольку работодатели не особенно рискуют нанимать на работу удаленно работающих разработчиков. Посетите местные семинары и неформальные встречи по интересам, свяжитесь с их участниками, и вам будет намного проще найти работу разработчика в области биткойна.

Аналогично, чтобы найти работу в удаленном режиме, нужно, чтобы вас заметили. Не только участвуйте в проектах с открытым кодом, но и посещайте конференции, устанавливайте контакты, создавайте и публикуйте техническое содержимое в Интернете (видеоролики в YouTube, блог-посты и т.д.). Все это поможет вам обратить на себя внимание и получить работу в удаленном режиме.

Заключение

Мне очень приятно, что вы дочитали эту книгу до конца. Если у вас появится желание поделиться своими успехами в освоении премудростей программирования биткойна, буду рад получить от вас известие! Пишите мне по адресу `jimmy@programmingblockchain.com`.

Ответы к упражнениям

Глава 1. Конечные поля

Упражнение 1

Напишите соответствующий метод `__ne__()`, в котором оба объекта типа `FieldElement` проверяются на *неравенство*.

```
class FieldElement:
...
  def __ne__(self, other):
    # этот метод должен выполнять операцию сравнения
    # на неравенство, обратную операции ==
    return not (self == other)
```

Упражнение 2

Решите приведенные ниже задачи в конечном поле F_{57} при условии, что все знаки + здесь обозначают операцию сложения $+_f$, а все знаки - — операцию вычитания $-_f$ в данном конечном поле.

- 44 + 33
- 9 - 29
- 17 + 42 + 49
- 52 - 30 - 38

```
>>> prime = 57
>>> print((44+33)%prime)
20
>>> print((9-29)%prime)
37
>>> print((17+42+49)%prime)
```

```
51
>>> print((52-30-38)%prime)
41
```

Упражнение 3

Напишите соответствующий метод `__sub__()`, в котором определяется операция вычитания двух объектов типа `FieldElement`.

```
class FieldElement:
...
  def __sub__(self, other):
    if self.prime != other.prime:
      raise TypeError(
            'Cannot subtract two numbers in different Fields')
    # поля self.num и other.num содержат конкретные значения
    # в поле self.prime находится значение для взятия по модулю
    num = (self.num - other.num) % self.prime
    # возвратить элемент того же самого класса
    return self.__class__(num, self.prime)
```

Упражнение 4

Решите приведенные ниже уравнения в конечном поле F_{97} при условии, что операции умножения и возведения в степень выполняются в данном конечном поле.

- $95 \times 45 \times 31$
- $17 \times 13 \times 19 \times 44$
- $12^7 \times 77^{49}$

```
>>> prime = 97
>>> print(95*45*31 % prime)
23
>>> print(17*13*19*44 % prime)
68
>>> print(12**7*77**49 % prime)
63
```

Упражнение 5

Какое из приведенных ниже множеств чисел находится в конечном поле F_{19}, если k = 1, 3, 7, 13, 18? Есть ли в этих множества что-нибудь примечательное?

$$\{k \times 0, k \times 1, k \times 2, k \times 3, \ldots k \times 18\}$$

```
>>> prime = 19
>>> for k in (1,3,7,13,18):
...     print([k*i % prime for i in range(prime)])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
[0, 3, 6, 9, 12, 15, 18, 2, 5, 8, 11, 14, 17, 1, 4, 7, 10, 13, 16]
[0, 7, 14, 2, 9, 16, 4, 11, 18, 6, 13, 1, 8, 15, 3, 10, 17, 5, 12]
[0, 13, 7, 1, 14, 8, 2, 15, 9, 3, 16, 10, 4, 17, 11, 5, 18, 12, 6]
[0, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> for k in (1,3,7,13,18):
...     print(sorted([k*i % prime for i in range(prime)]))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```

В результате сортировки всегда получается одно и то же множество.

Упражнение 6

Напишите соответствующий метод `__mul__()`, в котором определяется операция умножения двух элементов конечного поля.

```
class FieldElement:
...
  def __mul__(self, other):
    if self.prime != other.prime:
      raise TypeError(
            'Cannot multiply two numbers in different Fields')
    # поля self.num и other.num содержат конкретные значения
    # в поле self.prime находится значение для взятия по модулю
    num = (self.num * other.num) % self.prime
    # возвратить элемент того же самого класса
    return self.__class__(num, self.prime)
```

Упражнение 7

Если $p = 7, 11, 17, 31$, то какое из приведенных ниже множеств окажется в конечном поле F_p?

$$\{1^{(p-1)}, 2^{(p-1)}, 3^{(p-1)}, 4^{(p-1)}, \ldots (p-1)^{(p-1)}\}$$

```
>>> for prime in (7, 11, 17, 31):
...     print([pow(i, prime-1, prime) for i in range(1, prime)])
[1, 1, 1, 1, 1, 1]
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,\
 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Упражнение 8

Решите следующие задачи в конечном поле F_{31}:

- $3 / 24$
- 17^{-3}
- $4^{-4} \times 11$

```
>>> prime = 31
>>> print(3*pow(24, prime-2, prime) % prime)
4
>>> print(pow(17, prime-4, prime))
29
>>> print(pow(4, prime-5, prime)*11 % prime)
13
```

Упражнение 9

Напишите соответствующий метод `__truediv__()`, реализующий операцию деления двух элементов конечного поля.

Следует, однако, иметь в виду, что в версии Python 3 операция деления разделена по двум функциям — `__truediv__()` и `__floordiv__()`. Первая функция выполняет обычную операцию деления, а вторая — операцию целочисленного деления.

```
class FieldElement:
...
  def __truediv__(self, other):
    if self.prime != other.prime:
      raise TypeError(
            'Cannot divide two numbers in different Fields')
    # воспользоваться малой теоремой Ферма:
    # self.num**(p-1) % p == 1
    # это означает следующее:
    # 1/n == pow(n, p-2, p)
    # возвратить элемент того же самого класса
    num = self.num * pow(other.num, self.prime - 2, self.prime) \
          % self.prime
    return self.__class__(num, self.prime)
```

Глава 2. Эллиптические кривые

Упражнение 1

Выясните, какие из приведенных ниже точек находятся на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$.

(2,4), (–1,–1), (18,77), (5,7)

```
>>> def on_curve(x, y):
... return y**2 == x**3 + 5*x + 7
>>> print(on_curve(2,4))
False
>>> print(on_curve(-1,-1))
True
>>> print(on_curve(18,77))
True
>>> print(on_curve(5,7))
False
```

Упражнение 2

Напишите метод `__ne__()` для класса `Point`, чтобы сравнить в нем оба объекта данного класса на неравенство.

```
class Point:
...
  def __ne__(self, other):
    return not (self == other)
```

Упражнение 3

Реализуйте проверку в том случае, если обе точки являются аддитивной инверсией, т.е. имеют одинаковые координаты x, но разные координаты y, а следовательно, через них проходит вертикальная линия. В результате такой проверки должна быть возвращена бесконечно удаленная точка.

```
class Point:
...
    if self.x == other.x and self.y != other.y:
      return self.__class__(None, None, self.a, self.b)
```

Упражнение 4

Где на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$, находится точка, получаемая в результате сложения исходных точек с координатами (2,5) + (–1,–1)?

```
>>> x1, y1 = 2, 5
>>> x2, y2 = -1, -1
>>> s = (y2 - y1) / (x2 - x1)
>>> x3 = s**2 - x1 - x2
>>> y3 = s * (x1 - x3) - y1
>>> print(x3, y3)
3.0 -7.0
```

Упражнение 5

Напишите метод `__add__()` для проверки условия в том случае, когда $x_1 \neq x_2$.

```
class Point:
...
  def __add__(self, other):
    ...
    if self.x != other.x:
      s = (other.y - self.y) / (other.x - self.x)
      x = s**2 - self.x - other.x
      y = s * (self.x - x) - self.y
      return self.__class__(x, y, self.a, self.b)
```

Упражнение 6

Где на кривой, описываемой уравнением $y^2 = x^3 + 5x + 7$, находится точка, получаемая сложением исходных точек с координатами (–1,–1) + (–1,–1)?

```
>>> a, x1, y1 = 5, -1, -1
>>> s = (3 * x1**2 + a) / (2 * y1)
>>> x3 = s**2 - 2*x1
>>> y3 = s*(x1-x3)-y1
>>> print(x3,y3)
10.0 77.0
```

Упражнение 7

Напишите метод `__add__()` для проверки условия в том случае, когда $P_1 = P_2$.

```
class Point:
...
  def __add__(self, other):
    ...
    if self == other:
      s = (3 * self.x**2 + self.a) / (2 * self.y)
      x = s**2 - 2 * self.x
      y = s * (self.x - x) - self.y
      return self.__class__(x, y, self.a, self.b)
```

Глава 3. Криптография по эллиптическим кривым

Упражнение 1

Выясните, находятся ли приведенные ниже точки на кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

(192,105), (17,56), (200,119), (1,193), (42,99)

```
>>> from ecc import FieldElement
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> def on_curve(x,y):
...     return y**2 == x**3 + a*x + b
>>> print(on_curve(FieldElement(192, prime), FieldElement(105, prime)))
True
>>> print(on_curve(FieldElement(17, prime), FieldElement(56, prime)))
True
>>> print(on_curve(FieldElement(200, prime), FieldElement(119, prime)))
False
>>> print(on_curve(FieldElement(1, prime), FieldElement(193, prime)))
True
>>> print(on_curve(FieldElement(42, prime), FieldElement(99, prime)))
False
```

Упражнение 2

Выполните сложение приведенных ниже точек для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

- (170,142) + (60,139)
- (47,71) + (17,56)
- (143,98) + (76,66)

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> p1 = Point(FieldElement(170, prime), FieldElement(142, prime), a, b)
>>> p2 = Point(FieldElement(60, prime), FieldElement(139, prime), a, b)
>>> print(p1+p2)
Point(220,181)_0_7 FieldElement(223)
>>> p1 = Point(FieldElement(47, prime), FieldElement(71, prime), a, b)
>>> p2 = Point(FieldElement(17, prime), FieldElement(56, prime), a, b)
>>> print(p1+p2)
Point(215,68)_0_7 FieldElement(223)
>>> p1 = Point(FieldElement(143, prime), FieldElement(98, prime), a, b)
>>> p2 = Point(FieldElement(76, prime), FieldElement(66, prime), a, b)
>>> print(p1+p2)
Point(47,71)_0_7 FieldElement(223)
```

Упражнение 3

Выполните расширение класса ECCTest для проверки правильности операций сложения точек из предыдущего упражнения. Присвойте методу такой проверки имя test_add.

```
def test_add(self):
  prime = 223
  a = FieldElement(0, prime)
  b = FieldElement(7, prime)
  additions = (
    (192, 105, 17, 56, 170, 142),
    (47, 71, 117, 141, 60, 139),
    (143, 98, 76, 66, 47, 71),
  )
for x1_raw, y1_raw, x2_raw, y2_raw, x3_raw, y3_raw in additions:
  x1 = FieldElement(x1_raw, prime)
  y1 = FieldElement(y1_raw, prime)
  p1 = Point(x1, y1, a, b)
  x2 = FieldElement(x2_raw, prime)
  y2 = FieldElement(y2_raw, prime)
  p2 = Point(x2, y2, a, b)
  x3 = FieldElement(x3_raw, prime)
  y3 = FieldElement(y3_raw, prime)
  p3 = Point(x3, y3, a, b)
  self.assertEqual(p1 + p2, p3)
```

Упражнение 4

Выполните скалярное умножение приведенных ниже точек для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

- $2 \times (192,105)$
- $2 \times (143,98)$
- $2 \times (47,71)$
- $4 \times (47,71)$
- $8 \times (47,71)$
- $21 \times (47,71)$

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x1 = FieldElement(num=192, prime=prime)
>>> y1 = FieldElement(num=105, prime=prime)
>>> p = Point(x1,y1,a,b)
>>> print(p+p)
Point(49,71)_0_7 FieldElement(223)
>>> x1 = FieldElement(num=143, prime=prime)
>>> y1 = FieldElement(num=98, prime=prime)
>>> p = Point(x1,y1,a,b)
>>> print(p+p)
Point(64,168)_0_7 FieldElement(223)
>>> x1 = FieldElement(num=47, prime=prime)
>>> y1 = FieldElement(num=71, prime=prime)
>>> p = Point(x1,y1,a,b)
>>> print(p+p)
Point(36,111)_0_7 FieldElement(223)
>>> print(p+p+p+p)
Point(194,51)_0_7 FieldElement(223)
>>> print(p+p+p+p+p+p+p+p)
Point(116,55)_0_7 FieldElement(223)
>>> print(p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p+p)
Point(infinity)
```

Упражнение 5

Найдите порядок группы, сформированной из точки с координатами (15,86) для кривой, описываемой уравнением $y^2 = x^3 + 7$ над конечным полем F_{223}.

```
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x = FieldElement(15, prime)
>>> y = FieldElement(86, prime)
>>> p = Point(x, y, a, b)
>>> inf = Point(None, None, a, b)
>>> product = p
>>> count = 1
>>> while product != inf:
...     product += p
...     count += 1
>>> print(count)
7
```

Упражнение 6

Проверьте, действительны ли следующие подписи:

```
P = (0x887387e452b8eacc4acfde10d9aaf7f6d9a0f975aabb10d006e4da568744d06c,
     0x61de6d95231cd89026e286df3b6ae4a894a3378e393e93a0f45b666329a0ae34)

# подпись 1
z = 0xec208baa0fc1c19f708a9ca96fdeff3ac3f230bb4a7ba4aede4942ad003c0f60
r = 0xac8d1c87e51d0d441be8b3dd5b05c8795b48875dffe00b7ffcfac23010d3a395
s = 0x68342ceff8935ededd102dd876ffd6ba72d6a427a3edb13d26eb0781cb423c4

# подпись 2
z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d
r = 0xeff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c
s = 0xc7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6

>>> from ecc import S256Point, N, G
>>> point = S256Point(
...     0x887387e452b8eacc4acfde10d9aaf7f6d9a0f975aabb10d006\
       e4da568744d06c,
...     0x61de6d95231cd89026e286df3b6ae4a894a3378e393e93a0f45\
       b666329a0ae34)
>>> z = 0xec208baa0fc1c19f708a9ca96fdeff3ac3f230bb4a7ba4ae\
        de4942ad003c0f60
>>> r = 0xac8d1c87e51d0d441be8b3dd5b05c8795b48875dffe00b7ff\
cfac23010d3a395
>>> s = 0x68342ceff8935ededd102dd876ffd6ba72d6a427a3edb13d26\
eb0781cb423c4
>>> u = z * pow(s, N-2, N) % N
>>> v = r * pow(s, N-2, N) % N
>>> print((u*G + v*point).x.num == r)
True
```

```
>>> z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29\
307032997a838a3d
>>> r = 0xeff69ef2b1bd93a66ed5219add4fb51e11a840f404876325\
a1e8ffe0529a2c
>>> s = 0xc7207fee197d27c618aea621406f6bf5ef6fca38681d82b2\
f06fddbdce6feab6
>>> u = z * pow(s, N-2, N) % N
>>> v = r * pow(s, N-2, N) % N
>>> print((u*G + v*point).x.num == r)
True
```

Упражнение 7

Подпишите следующее сообщение заданным секретным ключом:

```
e = 12345
z = int.from_bytes(hash256('Programming Bitcoin!'), 'big')
```

```
>>> from ecc import S256Point, G, N
>>> from helper import hash256
>>> e = 12345
>>> z = int.from_bytes(hash256(b'Programming Bitcoin!'), 'big')
>>> k = 1234567890
>>> r = (k*G).x.num
>>> k_inv = pow(k, N-2, N)
>>> s = (z+r*e) * k_inv % N
>>> print(e*G)
S256Point(f01d6b9018ab421dd410404cb869072065522bf85734008f105\
cf385a023a80f, 0eba29d0f0c5408ed681984dc525982abefccd9f7ff01\
dd26da4999cf3f6a295)
>>> print(hex(z))
0x969f6056aa26f7d2795fd013fe88868d09c9f6aed96965016e1936ae47060d48
>>> print(hex(r))
0x2b698a0f0a4041b77e63488ad48c23e8e8838dd1fb7520408b121697b782ef22
>>> print(hex(s))
0x1dbc63bfef4416705e602a7b564161167076d8b20990a0f26f316cff2cb0bc1a
```

Глава 4. Сериализация

Упражнение 1

Представьте открытый ключ в несжатом формате SEC, если имеются следующие секретные ключи.

- 5000
- 2018^5
- 0xdeadbeef12345

```
>>> from ecc import PrivateKey
>>> priv = PrivateKey(5000)
>>> print(priv.point.sec(compressed=False).hex())
04ffe558e388852f0120e46af2d1b370f85854a8eb0841811ece0\
e3e03d282d57c315dc72890a4f10a1481c031b03b351b0dc79901\
ca18a00cf009dbdb157a1d10
>>> priv = PrivateKey(2018**5)
>>> print(priv.point.sec(compressed=False).hex())
04027f3da1918455e03c46f659266a1bb5204e959db7364d2\
f473bdf8f0a13cc9dff87647fd023c13b4a4994f17691895806\
e1b40b57f4fd22581a4f46851f3b06
>>> priv = PrivateKey(0xdeadbeef12345)
>>> print(priv.point.sec(compressed=False).hex())
04d90cd625ee87dd38656dd95cf79f65f60f7273b67d3096\
e68bd81e4f5342691f842efa762fd59961d0e99803c61edba8\
b3e3f7dc3a341836f97733aebf987121
```

Упражнение 2

Представьте открытый ключ в сжатом формате SEC, если имеются следующие секретные ключи.

- 5001
- 2019^5
- 0xdeadbeef54321

```
>>> from ecc import PrivateKey
>>> priv = PrivateKey(5001)
>>> print(priv.point.sec(compressed=True).hex())
0357a4f368868a8a6d572991e484e664810ff14c05c0fa023275251151fe0e53d1
>>> priv = PrivateKey(2019**5)
>>> print(priv.point.sec(compressed=True).hex())
02933ec2d2b111b92737ec12f1c5d20f3233a0ad21cd8b36d0bca7a0cfa5cb8701
>>> priv = PrivateKey(0xdeadbeef54321)
>>> print(priv.point.sec(compressed=True).hex())
0296be5b1292f6c856b3c5654e886fc13511462059089cdf9c479623bfcbe77690
```

Упражнение 3

Представьте в формате DER подпись со следующими величинами *r* и *s*:

```
r = 0x37206a0610995c58074999cb9767b87af4c4978db68c06e8e6e81d282047a7c6

s = 0x8ca63759c1157ebeaec0d03cecca119fc9a75bf8e6d0fa65c841c8e2738cdaec
```

```
>>> from ecc import Signature
>>> r = 0x37206a0610995c58074999cb9767b87af4c4978db68c06e8\
e6e81d282047a7c6
>>> s = 0x8ca63759c1157ebeaec0d03cecca119fc9a75bf8e6d0fa65\
c841c8e2738cdaec
>>> sig = Signature(r,s)
>>> print(sig.der().hex())
3045022037206a0610995c58074999cb9767b87af4c4978db68\
c06e8e6e81d282047a7c60221008ca63759c1157ebeaec0d03\
cecca119fc9a75bf8e6d0fa65c841c8e2738cdaec
```

Упражнение 4

Сначала преобразуйте приведенные ниже шестнадцатеричные значения в двоичную форму, а затем представьте их в кодировке Base58.

- `7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d`
- `eff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c`
- `c7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6`

```
>>> from helper import encode_base58
>>> h = '7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929\
e29307032997a838a3d'
>>> print(encode_base58(bytes.fromhex(h)))
9MA8fRQrT4u8Zj8ZRd6MAiiyaxb2Y1CMpvVkHQu5hVM6
>>> h = 'eff69ef2b1bd93a66ed5219add4fb51e11a840\
f404876325a1e8ffe0529a2c'
>>> print(encode_base58(bytes.fromhex(h)))
4fE3H2E6XMp4SsxtwinF7w9a34ooUrwWe4WsW1458Pd
>>> h = 'c7207fee197d27c618aea621406f6bf5ef6\
fca38681d82b2f06fddbdce6feab6'
>>> print(encode_base58(bytes.fromhex(h)))
EQJsjkd6JaGwxrjEhfeqPenqHwrBmPQZjJGNSCHBkcF7
```

Упражнение 5

Найдите адреса, соответствующие открытым ключам, которым отвечают следующие секретные ключи.

- 5002 (использовать несжатый формат SEC в сети testnet)
- 20205 (использовать сжатый формат SEC в сети testnet)
- 0x12345deadbeef (использовать сжатый формат SEC в сети mainnet)

```
>>> from ecc import PrivateKey
>>> priv = PrivateKey(5002)
>>> print(priv.point.address(compressed=False, testnet=True))
mmTPbXQFxboEtNRkwfh6K51jvdtHLxGeMA
>>> priv = PrivateKey(2020**5)
>>> print(priv.point.address(compressed=True, testnet=True))
mopVkxp8UhXqRYbCYJsbeE1h1fiF64jcoH
>>> priv = PrivateKey(0x12345deadbeef)
>>> print(priv.point.address(compressed=True, testnet=False))
1F1Pn2y6pDb68E5nYJJeba4TLg2U7B6KF1
```

Упражнение 6

Сформируйте секретный ключ в формате WIF, исходя из следующих секретных данных.

- 5003 (сжатый формат SEC для сети testnet)
- 2021^5 (несжатый формат SEC для сети testnet)
- 0x54321deadbeef (сжатый формат SEC для сети mainnet)

```
>>> from ecc import PrivateKey
>>> priv = PrivateKey(5003)
>>> print(priv.wif(compressed=True, testnet=True))
cMahea7zqjxrtgAbB7LSGbcQUr1uX1ojuat9jZodMN8rFTv2sfUK
>>> priv = PrivateKey(2021**5)
>>> print(priv.wif(compressed=False, testnet=True))
91avARGdfge8E4tZfYLoxeJ5sGBdNJQH4kvjpWAxgzczjbCwxic
>>> priv = PrivateKey(0x54321deadbeef)
>>> print(priv.wif(compressed=True, testnet=False))
KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgiuQJv1h8Ytr2S53a
```

Упражнение 7

Напишите функцию `little_endian_to_int()`, принимающую байты в том порядке, который принят в Python, интерпретирующую эти байты в прямом порядке следования и возвращающую целое число.

```
def little_endian_to_int(b):
    '''Этот метод принимает число, представленное
       последовательностью байтов в прямом порядке их
       следования, а возвращает целое число'''
    return int.from_bytes(b, 'little')
```

Упражнение 8

Напишите функцию `int_to_little_endian()`, выполняющую действия, противоположные функции из упражнения 7.

```
def int_to_little_endian(n, length):
    '''Этот метод принимает целое число и возвращает
       последовательность байтов в прямом порядке
       их следования, представляющую его длину'''
    return n.to_bytes(length, 'little')
```

Упражнение 9

Сформируйте самостоятельно адрес для сети testnet, используя самые длинные секретные данные, которые вам только известны. Это очень важно, поскольку в сети testnet действует немало ботов, пытающихся украсть циркулирующие в ней монеты. Непременно запишите на чем-нибудь эти секретные данные! Они вам еще пригодятся в дальнейшем для подписания транзакций.

```
>>> from ecc import PrivateKey
>>> from helper import hash256, little_endian_to_int
>>> passphrase = b'jimmy@programmingblockchain.com my secret'
>>> secret = little_endian_to_int(hash256(passphrase))
>>> priv = PrivateKey(secret)
>>> print(priv.point.address(testnet=True))
mft9LRNtaBNtpkknB8xgm17UvPedZ4ecYL
```

Глава 5. Транзакции

Упражнение 1

Напишите новую версию той части метода `parse()`, в которой производится синтаксический анализ транзакции. Чтобы сделать это надлежащим образом, вам придется преобразовать 4 байта в целое число, представленное байтами в прямом порядке их следования.

```
class Tx:
...
@classmethod
def parse(cls, s, testnet=False):
  version = little_endian_to_int(s.read(4))
  return cls(version, None, None, None, testnet=testnet)
```

Упражнение 2

Напишите ту часть метода `parse()` из класса `Tx` и метода `parse()` из класса `TxIn`, которая отвечает за синтаксический анализ вводов транзакции.

```
class Tx:
...
  @classmethod
  def parse(cls, s, testnet=False):
    version = little_endian_to_int(s.read(4))
    num_inputs = read_varint(s)
    inputs = []
    for _ in range(num_inputs):
      inputs.append(TxIn.parse(s))
    return cls(version, inputs, None, None, testnet=testnet)
...

class TxIn:
...
  @classmethod
  def parse(cls, s):
    '''Этот метод принимает поток байтов и сначала синтаксически
       анализирует ввод транзакции, а затем
       возвращает объект типа TxIn'''
    prev_tx = s.read(32)[::-1]
    prev_index = little_endian_to_int(s.read(4))
    script_sig = Script.parse(s)
    sequence = little_endian_to_int(s.read(4))
    return cls(prev_tx, prev_index, script_sig, sequence)
```

Упражнение 3

Напишите ту часть метода `parse()` из класса `Tx` и метода `parse()` из класса `TxOut`, которая отвечает за синтаксический анализ выводов транзакции.

```
class Tx:
...
  @classmethod
  def parse(cls, s, testnet=False):
    version = little_endian_to_int(s.read(4))
    num_inputs = read_varint(s)
    inputs = []
    for _ in range(num_inputs):
      inputs.append(TxIn.parse(s))
    num_outputs = read_varint(s)
    outputs = []
    for _ in range(num_outputs):
```

```
      outputs.append(TxOut.parse(s))
    return cls(version, inputs, outputs, None, testnet=testnet)
...

class TxOut:
...
  @classmethod
  def parse(cls, s):
    '''Этот метод принимает поток байтов и сначала синтаксически
       анализирует вывод транзакции, а затем
       возвращает объект типа TxOut'''
    amount = little_endian_to_int(s.read(8))
    script_pubkey = Script.parse(s)
    return cls(amount, script_pubkey)
```

Упражнение 4

Напишите ту часть метода `parse()` из класса `Tx`, которая отвечает за синтаксический анализ времени блокировки.

```
class Tx:
...
  @classmethod
  def parse(cls, s, testnet=False):
    version = little_endian_to_int(s.read(4))
    num_inputs = read_varint(s)
    inputs = []
    for _ in range(num_inputs):
      inputs.append(TxIn.parse(s))
    num_outputs = read_varint(s)
    outputs = []
    for _ in range(num_outputs):
      outputs.append(TxOut.parse(s))
    locktime = little_endian_to_int(s.read(4))
    return cls(version, inputs, outputs, locktime, testnet=testnet)
```

Упражнение 5

Что содержит поле ScriptSig на втором вводе, поле ScriptPubKey — на первом выводе и поле суммы — на втором выводе приведенной ниже транзакции?

```
010000000456919960ac691763688d3d3bcea9ad6ecaf875df5339e148\
a1fc61c6ed7a069e0100
00006a47304402204585bcdef85e6b1c6af5c2669d4830ff86e42dd205\
c0e089bc2a821657e951
c002201024a10366077f87d6bce1f7100ad8cfa8a064b39d4e8fe4ea13\
a7b71aa8180f012102f0
```

```
da57e85eec2934a82a585ea337ce2f4998b50ae699dd79f5880e253daf\
afb7feffffffeb8f51f4
038dc17e6313cf831d4f02281c2a468bde0fafd37f1bf882729e7fd\
3000000006a473044022078
99531a52d59a6de200179928ca900254a36b8dff8bb75f5f5d71b1cd\
c26125022008b422690b84
61cb52c3cc30330b23d574351872b7c361e9aae3649071c1a7160121035\
d5c93d9ac96881f19ba
1f686f15f009ded7c62efe85a872e6a19b43c15a2937feffffff567bf\
40595119d1bb8a3037c35
6efd56170b64cbcc160fb028fa10704b45d775000000006a47304402204c7\
c7818424c7f7911da
6cddc59655a70af1cb5eaf17c69dadbfc74ffa0b662f02207599e08bc\
8023693ad4e9527dc42c3
4210f7a7d1d1ddfc8492b654a11e7620a0012102158b46fbdff65d0172
\b7989aec8850aa0dae49
abfb84c81ae6e5b251a58ace5cfeffffffd63a5e6c16e620f86f375925\
b21cabaf736c779f88fd
04dcad51d26690f7f345010000006a47304402200633ea0d3314bea0d95\
b3cd8dadb2ef79ea833
1ffe1e61f762c0f6daea0fabde022029f23b3e9c30f080446150b2385202\
8751635dcee2be669c
2a1686a4b5edf304012103ffd6f4a67e94aba353a00882e563ff2722eb4cf\
f0ad6006e86ee20df
e7520d55feffffff0251430f00000000001976a914ab0c0b2e98b1ab6dbf\
67d4750b0a56244948
a87988ac005a6202000000001976a9143c82d7df364eb6c75be8c80df2\
b3eda8db57397088ac46430600

>>> from io import BytesIO
>>> from tx import Tx
>>> hex_transaction = '010000000456919960ac691763688d3d3bcea9ad\
6ecaf875df5339e148a1fc61c6ed7a069e010000006a47304402204585bcdef\
85e6b1c6af5c2669d4830ff86e42dd205c0e089bc2a821657e951c002201024\
a10366077f87d6bce1f7100ad8cfa8a064b39d4e8fe4ea13a7b71aa8180f0121\
02f0da57e85eec2934a82a585ea337ce2f4998b50ae699dd79f5880e253\
dafafb7feffffffeb8f51f4038dc17e6313cf831d4f02281c2a468bde0fafd37\
f1bf882729e7fd3000000006a47304402207899531a52d59a6de200179928ca9\
00254a36b8dff8bb75f5f5d71b1cdc26125022008b422690b8461cb52c3cc3033\
0b23d574351872b7c361e9aae3649071c1a7160121035d5c93d9ac96881f19ba1\
f686f15f009ded7c62efe85a872e6a19b43c15a2937feffffff567 bf40595119d1\
bb8a3037c356efd56170b64cbcc160fb028fa10704b45d775000000006a4730440\
2204c7c7818424c7f7911da6cddc59655a70af1cb5eaf17c69dadbfc74ffa0b662\
f02207599e08bc8023693ad4e9527dc42c34210f7a7d1d1ddfc8492b654a11e7620\
a0012102158b46fbdff65d0172b7989aec8850aa0dae49abfb84c81ae6e5b251a58\
ace5cfeffffffd63a5e6c16e620f86f375925b21cabaf736c779f88fd04dcad51d\
26690f7f345010000006a47304402200633ea0d3314bea0d95b3cd8dadb2ef79ea\
8331ffe1e61f762c0f6daea0fabde022029f23b3e9c30f080446150b2385202875\
```

```
1635dcee2be669c2a1686a4b5edf304012103ffd6f4a67e94aba353a00882e563ff\
2722eb4cff0ad6006e86ee20dfe7520d55feffffff0251430f00000000001976a
914\ab0c0b2e98b1ab6dbf67d4750b0a56244948a87988ac005a620200000000\
1976a9143c82d7df364eb6c75be8c80df2b3eda8db57397088ac46430600'
>>> stream = BytesIO(bytes.fromhex(hex_transaction))
>>> tx_obj = Tx.parse(stream)
>>> print(tx_obj.tx_ins[1].script_sig)
304402207899531a52d59a6de200179928ca900254a36b8dff8bb75f5f5d71b1
cdc26125022008b422690b8461cb52c3cc30330b23d574351872b7c361e9aae\
3649071c1a71601 035d5c93d9ac96881f19ba1f686f15f009ded7c62efe85a\
872e6a19b43c15a2937
>>> print(tx_obj.tx_outs[0].script_pubkey)
OP_DUP OP_HASH160 ab0c0b2e98b1ab6dbf67d4750b0a56244948a879 \
OP_EQUALVERIFY OP_CHECKSIG
>>> print(tx_obj.tx_outs[1].amount)
40000000
```

Упражнение 6

Напишите метод `fee()` для класса `Tx`, чтобы рассчитать плату за транзакцию.

```
class Tx:
...
  def fee(self, testnet=False):
    input_sum, output_sum = 0, 0
    for tx_in in self.tx_ins:
      input_sum += tx_in.value(testnet=testnet)
    for tx_out in self.tx_outs:
      output_sum += tx_out.amount
    return input_sum - output_sum
```

Глава 6. Язык Script

Упражнение 1

Напишите функцию `op_hash160()`.

```
def op_hash160(stack):
  if len(stack) < 1:
    return False
  element = stack.pop()
  h160 = hash160(element)
  stack.append(h160)
  return True
```

Упражнение 2

Напишите свою версию функции `op_checksig()` из исходного файла `op.py`.

```
def op_checksig(stack, z):
  if len(stack) < 2:
    return False
  sec_pubkey = stack.pop()
  der_signature = stack.pop()[:-1]
  try:
    point = S256Point.parse(sec_pubkey)
    sig = Signature.parse(der_signature)
  except (ValueError, SyntaxError) as e:
    return False
  if point.verify(z, sig):
    stack.append(encode_num(1))
  else:
    stack.append(encode_num(0))
  return True
```

Упражнение 3

Создайте сценарий ScriptSig, способный разблокировать открытый ключ по следующему сценарию ScriptPubKey:

```
767695935687
```

Имейте в виду, что в операции `OP_MUL` умножаются два верхних элемента в стеке, а коды в сценарии ScriptPubKey обозначают следующие операции.

- `56 = OP_6`
- `76 = OP_DUP`
- `87 = OP_EQUAL`
- `93 = OP_ADD`
- `95 = OP_MUL`

```
>>> from script import Script
>>> script_pubkey = Script([0x76, 0x76, 0x95, 0x93, 0x56, 0x87])
>>> script_sig = Script([0x52])
>>> combined_script = script_sig + script_pubkey
>>> print(combined_script.evaluate(0))
True
```

Операция `OP_2` с кодом 52 удовлетворит уравнению $x^2 + x - 6 = 0$.

Упражнение 4

Выясните назначение следующего сценария:

```
6e879169a77ca787
```

Коды в этом сценарии обозначают следующие операции.

- 69 = OP_VERIFY
- 6e = OP_2DUP
- 7c = OP_SWAP
- 87 = OP_EQUAL
- 91 = OP_NOT
- a7 = OP_SHA1

Воспользуйтесь методом `Script.parse()` и выясните назначение отдельных операций по их кодам, обратившись по адресу `https://en.bitcoin.it/wiki/Script`.

```
>>> from script import Script
>>> script_pubkey = Script([0x6e, 0x87, 0x91, 0x69, 0xa7,
                            0x7c, 0xa7, 0x87])
>>> c1 = '255044462d312e330a25e2e3cfd30a0a0a312030206f626a0a3c3\
c2f576964746820322030205220f4865696768742033203020522f5479706520\
34203020522f537562747970652035203020522f46696c7465722036203020522\
f436f6c6f725370616365203720302522f4c656e6774682038203020522f4269\
7473506572436f6d706f6e656e7420383e3e0a73747265616d0affd8fffe\
00245348412d3120697320646561642121212121852fec092339759c39\
b1a1c63c4c97e1fffe017f46dc93a6b67e013b029aaa1db2560b45ca67d688\
c7f84b8c4c791fe02b3df614f86db1690901c56b45c1530afedfb76038e972722\
fe7ad728f0e4904e046c230570fe9d41398abe12ef5bc942be33542a4802d98\
b5d70f2a332ec37fac3514e74ddc0f2cc1a874cd0c78305a21566461309789606\
bd0bf3f98cda8044629a1'
>>> c2 = '255044462d312e330a25e2e3cfd30a0a0a312030206f626a0a3c3c2\
f576964746820322030205220f4865696768742033203020522f5479706520342\
03020522f537562747970652035203020522f46696c746572203620302052 2f436\
f6c6f725370616365203720302522f4c656e6774682038203020522f42697473\
506572436f6d706f6e656e7420383e3e0a73747265616d0affd8fffe00245348412\
d3120697320646561642121212121852fec092339759c39b1a1c63c4c97e1fffe\
017346dc9166b67e118f029ab621b2560ff9ca67cca8c7f85ba84c79030c2b3de\
218f86db3a90901d5df45c14f26fedfb3dc38e96ac22fe7bd728f0e45bce046d23\
c570feb141398bb552ef5a0a82be331fea48037b8b5d71f0e332edf93ac3500eb4\
ddc0decc1a864790c782c76215660dd309791d06bd0af3f98cda4bc4629b1'
>>> collision1 = bytes.fromhex(c1)     ❶
>>> collision2 = bytes.fromhex(c2)
```

```
>>> script_sig = Script([collision1, collision2])
>>> combined_script = script_sig + script_pubkey
>>> print(combined_script.evaluate(0))
True
```

❶ Содержимое переменных `collision1` и `collision2` получено из прообразов по алгоритму SHA-1, для которых в компании Google обнаружены коллизии (`https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`).

В данном сценарии осуществляется поиск коллизии по алгоритму SHA-1. Единственный способ удовлетворить данный сценарий — предоставить координаты x и y (например, $x \neq y$, но `sha1(x) = sha1(y)`).

Глава 7. Создание и проверка достоверности транзакций

Упражнение 1

Напишите метод `sig_hash()` для класса `Tx`, чтобы реализовать в нем вычисление хеша подписи.

```
class Tx:
...
  def sig_hash(self, input_index):
    s = int_to_little_endian(self.version, 4)
    s += encode_varint(len(self.tx_ins))
    for i, tx_in in enumerate(self.tx_ins):
      if i == input_index:
        s += TxIn(
             prev_tx=tx_in.prev_tx,
             prev_index=tx_in.prev_index,
             script_sig=tx_in.script_pubkey(self.testnet),
             sequence=tx_in.sequence,
        ).serialize()
      else:
        s += TxIn(
             prev_tx=tx_in.prev_tx,
             prev_index=tx_in.prev_index,
             sequence=tx_in.sequence,
        ).serialize()
    s += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
      s += tx_out.serialize()
    s += int_to_little_endian(self.locktime, 4)
    s += int_to_little_endian(SIGHASH_ALL, 4)
```

```
    h256 = hash256(s)
    return int.from_bytes(h256, 'big')
```

Упражнение 2

Напишите метод `verify_input()` для класса `Tx`, чтобы реализовать в нем проверку ввода транзакции на достоверность. Для этого вам придется воспользоваться методами `TxIn.script_pubkey()`, `Tx.sig_hash()` и `Script.evaluate()`.

```
class Tx:
...
  def verify_input(self, input_index):
    tx_in = self.tx_ins[input_index]
    script_pubkey = tx_in.script_pubkey(testnet=self.testnet)
    z = self.sig_hash(input_index)
    combined = tx_in.script_sig + script_pubkey
    return combined.evaluate(z)
```

Упражнение 3

Напишите метод `sign_input()` для класса `Tx`, чтобы реализовать в нем подписание вводов транзакции.

```
class Tx:
...
  def sign_input(self, input_index, private_key):
    z = self.sig_hash(input_index)
    der = private_key.sign(z).der()
    sig = der + SIGHASH_ALL.to_bytes(1, 'big')
    sec = private_key.point.sec()
    self.tx_ins[input_index].script_sig = Script([sig, sec])
    return self.verify_input(input_index)
```

Упражнение 4

Создайте транзакцию для отправки 60% суммы из одного неизрасходованного вывода UTXO по адресу `mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv` в сети testnet. Оставшаяся сумма за вычетом платы за транзакцию должна вернуться обратно по вашему измененному адресу. Это должна быть транзакция с одним вводом и двумя выводами. Свою транзакцию вы можете переслать по адресу `https://live.blockcypher.com/btc/pushtx`.

```
>>> from ecc import PrivateKey
>>> from helper import decode_base58, SIGHASH_ALL
>>> from script import p2pkh_script, Script
>>> from tx import TxIn, TxOut, Tx
>>> prev_tx = bytes.fromhex('75a1c4bc671f55f626dda1074c7725991\
e6f68b8fcefcfca7b64405ca3b45f1c')
>>> prev_index = 1
>>> target_address = 'miKegze5FQNCnGw6PKyqUbYUeBa4x2hFeM'
>>> target_amount = 0.01
>>> change_address = 'mzx5YhAH9kNHtcN481u6WkjeHjYtVeKVh2'
>>> change_amount = 0.009
>>> secret = 8675309
>>> priv = PrivateKey(secret=secret)
>>> tx_ins = []
>>> tx_ins.append(TxIn(prev_tx, prev_index))
>>> tx_outs = []
>>> h160 = decode_base58(target_address)
>>> script_pubkey = p2pkh_script(h160)
>>> target_satoshis = int(target_amount*100000000)
>>> tx_outs.append(TxOut(target_satoshis, script_pubkey))
>>> h160 = decode_base58(change_address)
>>> script_pubkey = p2pkh_script(h160)
>>> change_satoshis = int(change_amount*100000000)
>>> tx_outs.append(TxOut(change_satoshis, script_pubkey))
>>> tx_obj = Tx(1, tx_ins, tx_outs, 0, testnet=True)
>>> print(tx_obj.sign_input(0, priv))
True
>>> print(tx_obj.serialize().hex())
01000000011c5fb4a35c40647bcacfeffcb8686f1e9925774c07a1dd26f6551\
f67bcc4a175010000006b483045022100a08ebb92422b3599a2d2fcdaa11f8f\
807a66ccf33e7f4a9ff0a3c51f1b1ec5dd02205ed21dfede5925362b8d9833\
e908646c54be7ac6664e31650159e8f69b6ca539012103935581e52c354cd2f\
484fe8ed83af7a3097005b2f9c60bff71d35bd795f54b67ffffffff024042\
0f00000000001976a9141ec51b3654c1f1d0f4929d11a1f702937eaf50c888\
ac9fbb0d00000000001976a914d52ad7ca9b3d096a38e752c2018e6fbc40\
cdf26f88ac00000000
```

Упражнение 5

Это упражнение посложнее. Приобретите немного монет из биткойнового крана в сети testnet и создайте транзакцию с двумя вводами и одним выводом. Один из вводов должен быть из биткойнового крана, другой — из предыдущего упражнения, а выводом может служить ваш собственный адрес. Свою транзакцию вы можете переслать по адресу `https://live.blockcypher.com/btc/pushtx`.

```
>>> from ecc import PrivateKey
>>> from helper import decode_base58, SIGHASH_ALL
>>> from script import p2pkh_script, Script
>>> from tx import TxIn, TxOut, Tx
>>> prev_tx_1 = bytes.fromhex('11d05ce707c1120248370d1cbf5561d\
22c4f83aeba0436792c82e0bd57fe2a2f')
>>> prev_index_1 = 1
>>> prev_tx_2 = bytes.fromhex('51f61f77bd061b9a0da60d4bedaaf1b1\
fad0c11e65fdc744797ee22d20b03d15')
>>> prev_index_2 = 1
>>> target_address = 'mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv'
>>> target_amount = 0.0429
>>> secret = 8675309
>>> priv = PrivateKey(secret=secret)
>>> tx_ins = []
>>> tx_ins.append(TxIn(prev_tx_1, prev_index_1))
>>> tx_ins.append(TxIn(prev_tx_2, prev_index_2))
>>> tx_outs = []
>>> h160 = decode_base58(target_address)
>>> script_pubkey = p2pkh_script(h160)
>>> target_satoshis = int(target_amount*100000000)
>>> tx_outs.append(TxOut(target_satoshis, script_pubkey))
>>> tx_obj = Tx(1, tx_ins, tx_outs, 0, testnet=True)
>>> print(tx_obj.sign_input(0, priv))
True
>>> print(tx_obj.sign_input(1, priv))
True
>>> print(tx_obj.serialize().hex())
01000000022f2afe57bde0822c793604baae834f2cd26155bf1c0d37480212\
c107e75cd011010000006a47304402204cc5fe11b2b025f8fc9f6073b5e\
3942883bbba266b71751068badeb8f11f0364022070178363f5dea4149581\
a4b9b9dbad91ec1fd990e3fa14f9de3ccb421fa5b269012103935581\
e52c354cd2f484fe8ed83af7a3097005b2f9c60bff71d35bd795f54b67\
ffffffff153db0202de27e7944c7fd651ec1d0fab1f1aaed4b0da60d9a1b\
06bd771ff651010000006b483045022100b7a938d4679aa7271f0d32d83b\
61a85eb0180cf1261d44feaad23dfd9799dafb02205ff2f366ddd9555\
f7146861a8298b7636be8b292090a224c5dc84268480d8be1012103935581\
e52c354cd2f484fe8ed83af7a3097005b2f9c60bff71d35bd795f54b67\
ffffffff01d075410000000001976a914ad346f8eb57dee9a37981716\
e498120ae80e44f788ac00000000
```

Глава 8. Оплата по хешу сценария

Упражнение 1

Напишите окончательный вариант функции `op_checkmultisig()` из исходного файла `op.py`.

```
def op_checkmultisig(stack, z):
  if len(stack) < 1:
    return False
  n = decode_num(stack.pop())
  if len(stack) < n + 1:
    return False
  sec_pubkeys = []
  for _ in range(n):
    sec_pubkeys.append(stack.pop())
  m = decode_num(stack.pop())
  if len(stack) < m + 1:
    return False
  der_signatures = []
  for _ in range(m):
    der_signatures.append(stack.pop()[:-1])
  stack.pop()
  try:
    points = [S256Point.parse(sec) for sec in sec_pubkeys]
    sigs = [Signature.parse(der) for der in der_signatures]
    for sig in sigs:
      if len(points) == 0:
        return False
      while points:
        point = points.pop(0)
        if point.verify(z, sig):
          break
    stack.append(encode_num(1))
  except (ValueError, SyntaxError):
    return False
  return True
```

Упражнение 2

Напишите функцию `h160_to_p2pkh_address()`, преобразующую 20-байтовый хеш-код hash160 в адрес p2pkh.

```
def h160_to_p2pkh_address(h160, testnet=False):
  if testnet:
    prefix = b'\x6f'
  else:
```

```
    prefix = b'\x00'
  return encode_base58_checksum(prefix + h160)
```

Упражнение 3

Напишите функцию `h160_to_p2sh_address()`, преобразующую 20-байтовый хеш-код hash160 в адрес p2sh.

```
def h160_to_p2sh_address(h160, testnet=False):
  if testnet:
    prefix = b'\xc4'
  else:
    prefix = b'\x05'
  return encode_base58_checksum(prefix + h160)
```

Упражнение 4

Проверьте на достоверность вторую подпись из предыдущей транзакции.

```
>>> from io import BytesIO
>>> from ecc import S256Point, Signature
>>> from helper import hash256, int_to_little_endian
>>> from script import Script
>>> from tx import Tx, SIGHASH_ALL
>>> hex_tx = '0100000001868278ed6ddfb6c1ed3ad5f8181eb0c7a385\
aa0836f01d5e4789e6bd304d87221a000000db00483045022100dc92655fe\
37036f47756db8102e0d7d5e28b3beb83a8fef4f5dc0559bddfb94e02205\
a36d4e4e6c7fcd16658c50783e00c341609977aed3ad00937bf4ee942a899\
3701483045022100da6bee3c93766232079a01639d07fa869598749729ae\
323eab8eef53577d611b02207bef15429dcadce2121ea07f233115c6f09034\
c0be68db99980b9a6c5e754022014752210226626e955ea6ea6d98850c994f\
9107b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030\
a0e9feed096bed8045c8b98bec453e1ffac7fbdbd4bb7152aeffffffff04\
d3b11400000000001976a914904a49878c0adfc3aa05de7afad2cc15f483\
a56a88ac7f400900000000001976a914418327e3f3dda4cf5b9089325a4b\
95abdfa0334088ac722c0c00000000001976a914ba35042cfe9fc66fd35ac\
2224eebdafd1028ad2788acdc4ace020000000017a91474d691da1574\
e6b3c192ecfb52cc8984ee7b6c568700000000'
>>> hex_sec = '03b287eaf122eea69030a0e9feed096bed8045c8b98bec\
453e1ffac7fbdbd4bb71'
>>> hex_der = '3045022100da6bee3c93766232079a01639d07fa86959874\
9729ae323eab8eef53577d611b02207bef15429dcadce2121ea07f233115c6f\
09034c0be68db99980b9a6c5e754022'
>>> hex_redeem_script = '475221022626e955ea6ea6d98850c994f9107\
b036b1334f18ca8830bfff1295d21cfdb702103b287eaf122eea69030a0e9feed\
096bed8045c8b98bec453e1ffac7fbdbd4bb7152ae'
>>> sec = bytes.fromhex(hex_sec)
```

```
>>> der = bytes.fromhex(hex_der)
>>> redeem_script = Script.parse(
                    BytesIO(bytes.fromhex(hex_redeem_script)))
>>> stream = BytesIO(bytes.fromhex(hex_tx))
>>> tx_obj = Tx.parse(stream)
>>> s = int_to_little_endian(tx_obj.version, 4)
>>> s += encode_varint(len(tx_obj.tx_ins))
>>> i = tx_obj.tx_ins[0]
>>> s += TxIn(i.prev_tx, i.prev_index, redeem_script, i.sequence)
              .serialize()
>>> s += encode_varint(len(tx_obj.tx_outs))
>>> for tx_out in tx_obj.tx_outs:
... s += tx_out.serialize()
>>> s += int_to_little_endian(tx_obj.locktime, 4)
>>> s += int_to_little_endian(SIGHASH_ALL, 4)
>>> z = int.from_bytes(hash256(s), 'big')
>>> point = S256Point.parse(sec)
>>> sig = Signature.parse(der)
>>> print(point.verify(z, sig))
True
```

Упражнение 5

Видоизмените методы `sig_hash()` и `verify_input()` таким образом, чтобы производить верификацию транзакций в p2sh.

```
class Tx:
...
  def sig_hash(self, input_index, redeem_script=None):
    '''Возвращает целочисленное представление хеша, которое
       требуется получить со знаком для индекса input_index'''
    s = int_to_little_endian(self.version, 4)
    s += encode_varint(len(self.tx_ins))
    for i, tx_in in enumerate(self.tx_ins):
      if i == input_index:
        if redeem_script:
          script_sig = redeem_script
        else:
          script_sig = tx_in.script_pubkey(self.testnet)
      else:
        script_sig = None
      s += TxIn(
          prev_tx=tx_in.prev_tx,
          prev_index=tx_in.prev_index,
          script_sig=script_sig,
          sequence=tx_in.sequence,
      ).serialize()
```

```
    s += encode_varint(len(self.tx_outs))
    for tx_out in self.tx_outs:
      s += tx_out.serialize()
    s += int_to_little_endian(self.locktime, 4)
    s += int_to_little_endian(SIGHASH_ALL, 4)
    h256 = hash256(s)
    return int.from_bytes(h256, 'big')

  def verify_input(self, input_index):
    tx_in = self.tx_ins[input_index]
    script_pubkey = tx_in.script_pubkey(testnet=self.testnet)
    if script_pubkey.is_p2sh_script_pubkey():
      cmd = tx_in.script_sig.cmds[-1]
      raw_redeem = encode_varint(len(cmd)) + cmd
      redeem_script = Script.parse(BytesIO(raw_redeem))
    else:
      redeem_script = None
    z = self.sig_hash(input_index, redeem_script)
    combined = tx_in.script_sig + script_pubkey
    return combined.evaluate(z)
```

Глава 9. Блоки

Упражнение 1

Напишите метод `is_coinbase()` для класса `Tx`, в котором определяется, является ли транзакция монетизирующей.

```
class Tx:
...
  def is_coinbase(self):
    if len(self.tx_ins) != 1:
      return False
    first_input = self.tx_ins[0]
    if first_input.prev_tx != b'\x00' * 32:
      return False
    if first_input.prev_index != 0xffffffff:
      return False
    return True
```

Упражнение 2

Напишите метод `coinbase_height()` для класса `Tx`, в котором определяется высота блока монетизирующей транзакции.

```
class Tx:
...
  def coinbase_height(self):
    if not self.is_coinbase():
      return None
    element = self.tx_ins[0].script_sig.cmds[0]
    return little_endian_to_int(element)
```

Упражнение 3

Напишите метод `parse()` для класса `Block`, чтобы реализовать в нем синтаксический анализ блока.

```
class Block:
...
  @classmethod
  def parse(cls, s):
    version = little_endian_to_int(s.read(4))
    prev_block = s.read(32)[::-1]
    merkle_root = s.read(32)[::-1]
    timestamp = little_endian_to_int(s.read(4))
    bits = s.read(4)
    nonce = s.read(4)
    return cls(version, prev_block, merkle_root,
               timestamp, bits, nonce)
```

Упражнение 4

Напишите метод `serialize()` для класса `Block`, чтобы реализовать в нем сериализацию блока.

```
class Block:
...
  def serialize(self):
    result = int_to_little_endian(self.version, 4)
    result += self.prev_block[::-1]
    result += self.merkle_root[::-1]
    result += int_to_little_endian(self.timestamp, 4)
    result += self.bits
    result += self.nonce
    return result
```

Упражнение 5

Напишите метод `hash()` для класса `Block`, чтобы реализовать в нем хеширование блока.

```
class Block:
...
  def hash(self):
    s = self.serialize()
    sha = hash256(s)
    return sha[::-1]
```

Упражнение 6

Напишите метод `bip9()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP0009.

```
class Block:
...
  def bip9(self):
    return self.version >> 29 == 0b001
```

Упражнение 7

Напишите метод `bip91()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP00091.

```
class Block:
...
  def bip91(self):
    return self.version >> 4 & 1 == 1
```

Упражнение 8

Напишите метод `bip141()` для класса `Block`, чтобы реализовать проверку номера версии, соответствующего протоколу BIP0141.

```
class Block:
...
  def bip141(self):
    return self.version >> 1 & 1 == 1
```

Упражнение 9

Напишите функцию `bits_to_target()` из исходного файла `helper.py`, чтобы вычислить цель из заданных битов.

```
def bits_to_target(bits):
  exponent = bits[-1]
  coefficient = little_endian_to_int(bits[:-1])
  return coefficient * 256**(exponent - 3)
```

Упражнение 10

Напишите метод `difficulty()` для класса `Block`, чтобы определить в нем сложность добычи.

```
class Block:
...
  def difficulty(self):
    lowest = 0xffff * 256**(0x1d - 3)
    return lowest / self.target()
```

Упражнение 11

Напишите метод `check_pow()` для класса `Block`, чтобы проверить в нем подтверждение работы.

```
class Block:
...
  def check_pow(self):
    sha = hash256(self.serialize())
    proof = little_endian_to_int(sha)
    return proof < self.target()
```

Упражнение 12

Вычислите новые биты при условии, что первый и последний из 2016 блоков в период корректировки сложности таковы.

- Блок 471744

```
000000203471101bbda3fe307664b3283a9ef0e97d9a38a7eacd88\
00000000000000000010c8aba8479bbaa5e0848152fd3c2289ca50\
e1c3e58c9a4faaafbdf5803c5448ddb845597e8b0118e43a81d3
```

- Блок 473759

```
02000020f1472d9db4b563c35f97c428ac903f23b7fc055d1cfc26\
0000000000000000000b3f449fcbe1bc4cfbcb8283a0d2c037f961\
a3fdf2b8bedc144973735eea707e1264258597e8b0118e5f00474
```

```
>>> from io import BytesIO
>>> from block import Block
>>> from helper import TWO_WEEKS
>>> from helper import target_to_bits
>>> block1_hex = '000000203471101bbda3fe307664b3283a9ef0e97d9\
a38a7eacd8800000000000000000000010c8aba8479bbaa5e0848152fd3c2289\
ca50e1c3e58c9a4faaafbdf5803c5448ddb845597e8b0118e43a81d3'
>>> block2_hex = '02000020f1472d9db4b563c35f97c428ac903f23b7fc\
```

```
055d1cfc260000000000000000b3f449fcbe1bc4cfbcb8283a0d2c037f961\
a3fdf2b8bedc144973735eea707e1264258597e8b0118e5f00474'
>>> last_block = Block.parse(BytesIO(bytes.fromhex(block1_hex)))
>>> first_block = Block.parse(BytesIO(bytes.fromhex(block2_hex)))
>>> time_differential = last_block.timestamp - first_block.timestamp
>>> if time_differential > TWO_WEEKS * 4:
...     time_differential = TWO_WEEKS * 4
>>> if time_differential < TWO_WEEKS // 4:
...     time_differential = TWO_WEEKS // 4
>>> new_target = last_block.target()
                 * time_differential // TWO_WEEKS
>>> new_bits = target_to_bits(new_target)
>>> print(new_bits.hex())
80df6217
```

Упражнение 13

Напишите функцию `calculate_new_bits()` из исходного файла `helper.py` для вычисления новых битов.

```
def calculate_new_bits(previous_bits, time_differential):
if time_differential > TWO_WEEKS * 4:
time_differential = TWO_WEEKS * 4
if time_differential < TWO_WEEKS // 4:
time_differential = TWO_WEEKS // 4
new_target = bits_to_target(previous_bits)
             * time_differential // TWO_WEEKS
return target_to_bits(new_target)
```

Глава 10. Организация сети

Упражнение 1

Напишите метод `parse()` для класса `NetworkEnvelope`, чтобы реализовать синтаксический анализ сетевого сообщения.

```
@classmethod
def parse(cls, s, testnet=False):
  magic = s.read(4)
  if magic == b'':
    raise IOError('Connection reset!')
  if testnet:
    expected_magic = TESTNET_NETWORK_MAGIC
  else:
    expected_magic = NETWORK_MAGIC
  if magic != expected_magic:
```

```
    raise SyntaxError('magic is not right {} vs {}'
                      .format(magic.hex(), expected_magic.hex()))
  command = s.read(12)
  command = command.strip(b'\x00')
  payload_length = little_endian_to_int(s.read(4))
  checksum = s.read(4)
  payload = s.read(payload_length)
  calculated_checksum = hash256(payload)[:4]
  if calculated_checksum != checksum:
    raise IOError('checksum does not match')
  return cls(command, payload, testnet=testnet)
```

Упражнение 2

Выясните, что именно содержит следующее сетевое сообщение:

```
f9beb4d976657261636b000000000000000000005df6e0e2
```

```
class NetworkEnvelope:
...
   >>> from network import NetworkEnvelope
   >>> from io import BytesIO
   >>> message_hex =
       `f9beb4d976657261636b000000000000000000005df6e0e2`
   >>> stream = BytesIO(bytes.fromhex(message_hex))
   >>> envelope = NetworkEnvelope.parse(stream)
   >>> print(envelope.command)
   b'verack'
   >>> print(envelope.payload)
   b''
```

Упражнение 3

Напишите метод `serialize()` для класса `NetworkEnvelope`, чтобы реализовать сериализацию содержимого сетевого сообщения.

```
class NetworkEnvelope:
...
  def serialize(self):
    result = self.magic
    result += self.command + b'\x00' * (12 - len(self.command))
    result += int_to_little_endian(len(self.payload), 4)
    result += hash256(self.payload)[:4]
    result += self.payload
    return result
```

Упражнение 4

Напишите метод `serialize()` для класса `VersionMessage`, чтобы реализовать в нем сериализацию сообщения о версии сетевого протокола.

```
class VersionMessage:
...
  def serialize(self):
    result = int_to_little_endian(self.version, 4)
    result += int_to_little_endian(self.services, 8)
    result += int_to_little_endian(self.timestamp, 8)
    result += int_to_little_endian(self.receiver_services, 8)
    result += b'\x00' * 10 + b'\xff\xff' + self.receiver_ip
    result += int_to_little_endian(self.receiver_port, 2)
    result += int_to_little_endian(self.sender_services, 8)
    result += b'\x00' * 10 + b'\xff\xff' + self.sender_ip
    result += int_to_little_endian(self.sender_port, 2)
    result += self.nonce
    result += encode_varint(len(self.user_agent))
    result += self.user_agent
    result += int_to_little_endian(self.latest_block, 4)
    if self.relay:
      result += b'\x01'
    else:
      result += b'\x00'
    return result
```

Упражнение 5

Напишите метод `handshake()` для класса `SimpleNode`, чтобы реализовать в нем подтверждение подключения к сети.

```
class SimpleNode:
...
  def handshake(self):
    version = VersionMessage()
    self.send(version)
    self.wait_for(VerAckMessage)
```

Упражнение 6

Напишите метод `serialize()` для класса `GetHeadersMessage`, чтобы реализовать в нем сериализацию сообщения о получении заголовков блоков.

```
class GetHeadersMessage:
...
  def serialize(self):
```

```
    result = int_to_little_endian(self.version, 4)
    result += encode_varint(self.num_hashes)
    result += self.start_block[::-1]
    result += self.end_block[::-1]
    return result
```

Глава 11. Упрощенная проверка оплаты

Упражнение 1

Напишите функцию `merkle_parent()`, реализующую получение родительского хеша в дереве Меркла.

```
def merkle_parent(hash1, hash2):
  '''Этот метод принимает двоичные хеши и вычисляет
     хеш-код hash256'''
  return hash256(hash1 + hash2)
```

Упражнение 2

Напишите функцию `merkle_parent_level()`, вычисляющую родительский уровень дерева Меркла.

```
def merkle_parent_level(hashes):
  '''Этот метод принимает список двоичных хешей и возвращает
     вдвое меньший по длине список'''
  if len(hashes) == 1:
    raise RuntimeError(
          'Cannot take a parent level with only 1 item')
  if len(hashes) % 2 == 1:
    hashes.append(hashes[-1])
  parent_level = []
  for i in range(0, len(hashes), 2):
    parent = merkle_parent(hashes[i], hashes[i + 1])
    parent_level.append(parent)
  return parent_level
```

Упражнение 3

Напишите функцию `merkle_root()`, вычисляющую корень дерева Меркла.

```
def merkle_root(hashes):
  '''Этот метод принимает список двоичных хешей и возвращает
     корень дерева Меркла'''
  current_level = hashes
  while len(current_level) > 1:
```

```
    current_level = merkle_parent_level(current_level)
  return current_level[0]
```

Упражнение 4

Напишите метод `validate_merkle_root()` для класса `Block`, чтобы реализовать в нем проверку корня дерева Меркла на достоверность.

```
class Block:
...
  def validate_merkle_root(self):
    hashes = [h[::-1] for h in self.tx_hashes]
    root = merkle_root(hashes)
    return root[::-1] == self.merkle_root
```

Упражнение 5

Создайте пустое дерево Меркла с 27 узлами и выведите на экран каждый его уровень.

```
>>> import math
>>> total = 27
>>> max_depth = math.ceil(math.log(total, 2))
>>> merkle_tree = []
>>> for depth in range(max_depth + 1):
...   num_items = math.ceil(total / 2**(max_depth - depth))
...   level_hashes = [None] * num_items
...   merkle_tree.append(level_hashes)
>>> for level in merkle_tree:
...    print(level)
[None]
[None, None]
[None, None, None, None]
[None, None, None, None, None, None, None]
[None, None, None, None, None, None, None, None, None, None,
 None, None, None, None]
[None, None, None, None, None, None, None, None, None, None,
 None, None, None, None, None, None, None, None, None, None,
 None, None, None, None, None, None, None]
```

Упражнение 6

Напишите метод `parse()` для класса `MerkleBlock`, в котором синтактически анализируется сетевая команда `merkleblock`.

```
class MerkleBlock:
...
  @classmethod
  def parse(cls, s):
    version = little_endian_to_int(s.read(4))
    prev_block = s.read(32)[::-1]
    merkle_root = s.read(32)[::-1]
    timestamp = little_endian_to_int(s.read(4))
    bits = s.read(4)
    nonce = s.read(4)
    total = little_endian_to_int(s.read(4))
    num_hashes = read_varint(s)
    hashes = []
    for _ in range(num_hashes):
      hashes.append(s.read(32)[::-1])
    flags_length = read_varint(s)
    flags = s.read(flags_length)
    return cls(version, prev_block, merkle_root, timestamp,
               bits, nonce, total, hashes, flags)
```

Упражнение 7

Напишите метод `is_valid()` для класса `MerkleBlock`, чтобы реализовать в нем проверку древовидного блока Меркла на достоверность.

```
class MerkleBlock:
...
  def is_valid(self):
    flag_bits = bytes_to_bit_field(self.flags)
    hashes = [h[::-1] for h in self.hashes]
    merkle_tree = MerkleTree(self.total)
    merkle_tree.populate_tree(flag_bits, hashes)
    return merkle_tree.root()[::-1] == self.merkle_root
```

Глава 12. Фильтры Блума

Упражнение 1

Рассчитайте фильтр Блума для строк "Hello world" и "Goodbye", применив хеш-функцию hash160 к битовому полю размером 10.

```
>>> from helper import hash160
>>> bit_field_size = 10
>>> bit_field = [0] * bit_field_size
>>> for item in (b'hello world', b'goodbye'):
...     h = hash160(item)
```

```
...     bit = int.from_bytes(h, 'big') % bit_field_size
...     bit_field[bit] = 1
>>> print(bit_field)
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

Упражнение 2

Если задан фильтр Блума с параметрами `size=10`, `function_count=5`, `tweak=99`, то какие байты устанавливаются после ввода приведенных ниже элементов? (*Подсказка:* воспользуйтесь функцией `bit_field_to_bytes()` из исходного файла `helper.py` для преобразования содержимого битового поля в байты.)

- `b'Hello World'`
- `b'Goodbye!'`

```
>>> from bloomfilter import BloomFilter, BIP37_CONSTANT
>>> from helper import bit_field_to_bytes, murmur3
>>> field_size = 10
>>> function_count = 5
>>> tweak = 99
>>> items = (b'Hello World', b'Goodbye!')
>>> bit_field_size = field_size * 8
>>> bit_field = [0] * bit_field_size
>>> for item in items:
...     for i in range(function_count):
...       seed = i * BIP37_CONSTANT + tweak
...       h = murmur3(item, seed=seed)
...       bit = h % bit_field_size
...       bit_field[bit] = 1
>>> print(bit_field_to_bytes(bit_field).hex())
4000600a080000010940
```

Упражнение 3

Напишите метод `add()` для класса `BloomFilter`, чтобы реализовать в нем ввод элементов в фильтр Блума.

```
class BloomFilter:
...
  def add(self, item):
    for i in range(self.function_count):
      seed = i * BIP37_CONSTANT + self.tweak
      h = murmur3(item, seed=seed)
      bit = h % (self.size * 8)
      self.bit_field[bit] = 1
```

Упражнение 4

Напишите метод `filterload()` для класса `BloomFilter`, чтобы реализовать в нем загрузку фильтра Блума.

```
class BloomFilter:
...
  def filterload(self, flag=1):
    payload = encode_varint(self.size)
    payload += self.filter_bytes()
    payload += int_to_little_endian(self.function_count, 4)
    payload += int_to_little_endian(self.tweak, 4)
    payload += int_to_little_endian(flag, 1)
    return GenericMessage(b'filterload', payload)
```

Упражнение 5

Напишите метод `serialize()` для класса `GetDataMessage`, чтобы реализовать в нем сериализацию получаемых элементов данных.

```
class GetDataMessage:
...
  def serialize(self):
    result = encode_varint(len(self.data))
    for data_type, identifier in self.data:
      result += int_to_little_endian(data_type, 4)
      result += identifier[::-1]
    return result
```

Упражнение 6

Получите идентификатор текущего блока в сети testnet, отправьте себе немного монет по сети testnet, найдите неизрасходованный вывод транзакции UTXO, соответствующий этим монетам, *не* пользуясь обозревателем блоков, создайте транзакцию, используя данный UTXO в качестве ввода, а также перешлите сообщение `tx` по сети testnet.

```
>>> import time
>>> from block import Block
>>> from bloomfilter import BloomFilter
>>> from ecc import PrivateKey
>>> from helper import (
...     decode_base58,
...     encode_varint,
...     hash256,
...     little_endian_to_int,
```

```
...     read_varint,
... )
>>> from merkleblock import MerkleBlock
>>> from network import (
...     GetDataMessage,
...     GetHeadersMessage,
...     HeadersMessage,
...     NetworkEnvelope,
...     SimpleNode,
...     TX_DATA_TYPE,
...     FILTERED_BLOCK_DATA_TYPE,
... )
>>> from script import p2pkh_script, Script
>>> from tx import Tx, TxIn, TxOut
>>> last_block_hex = '00000000000000a03f9432ac63813c6710bfe41712\
ac5ef6faab093fe2917636'
>>> secret = little_endian_to_int(hash256(b'Jimmy Song'))
>>> private_key = PrivateKey(secret=secret)
>>> addr = private_key.point.address(testnet=True)
>>> h160 = decode_base58(addr)
>>> target_address = 'mwJn1YPMq7y5F8J3LkC5Hxg9PHyZ5K4cFv'
>>> target_h160 = decode_base58(target_address)
>>> target_script = p2pkh_script(target_h160)
>>> fee = 5000 # взимаемая плата в сатоши
>>> # подключиться к узлу testnet.programmingbitcoin.com
    # в режиме работы сети testnet
>>> node = SimpleNode('testnet.programmingbitcoin.com', \
                      testnet=True, logging=False)
>>> # создать фильтр Блума размером 30 и 5 хеш-функций,
    # а также добавить настройку
>>> bf = BloomFilter(30, 5, 90210)
>>> # ввести хеш-код h160 в фильтр Блума
>>> bf.add(h160)
>>> # завершить процедуру подтверждения установления связи
>>> node.handshake()
>>> # загрузить фильтр Блума по команде filterload
>>> node.send(bf.filterload())
>>> # установить упомянутый выше блок last_block в
    # качестве начального блока
>>> start_block = bytes.fromhex(last_block_hex)
>>> # отправить сообщение с начальным блоком по команде getheaders
>>> getheaders = GetHeadersMessage(start_block=start_block)
>>> node.send(getheaders)
>>> # ожидать сообщения с заголовками блоков
>>> headers = node.wait_for(HeadersMessage)
>>> # сохранить последний блок как None
>>> last_block = None
>>> # инициализировать объект типа GetDataMessage
```

```
>>> getdata = GetDataMessage()
>>> # перебрать заголовки блоков в цикле
>>> for b in headers.blocks:
... # проверить подтверждение работы в блоке на достоверность
... if not b.check_pow():
... raise RuntimeError('proof of work is invalid')
... # проверить, является ли предыдущий блок последним блоком
... if last_block is not None and b.prev_block != last_block:
... raise RuntimeError('chain broken')
... # добавить новый элемент в сообщение для получения данных,
... # он должен быть типа FILTERED_BLOCK_DATA_TYPE и хешем блока
... getdata.add_data(FILTERED_BLOCK_DATA_TYPE, b.hash())
... # установить в последнем блоке текущий хеш
... last_block = b.hash()
>>> # отправить сообщение для получения данных
>>> node.send(getdata)
>>> # инициализировать поля prev_tx, prev_index и prev_amount
    # значением None
>>> prev_tx, prev_index, prev_amount = None, None, None
>>> # выполнять цикл while до тех пор, пока prev_tx = None
>>> while prev_tx is None:
...   # ожидать блок из дерева Меркла или команды tx
...   message = node.wait_for(MerkleBlock, Tx)
...   # если имеется команда merkleblock,
...   if message.command == b'merkleblock':
...     # проверить объект типа MerkleBlock на достоверность,
...     if not message.is_valid():
...       raise RuntimeError('invalid merkle proof')
...       # иначе имеется команда tx
...   else:
...     # установить логическое значение True, обозначающее
...     # режим обработки транзакций в сети testnet
...     message.testnet = True
...     # циклически обработать выводы транзакции
...     for i, tx_out in enumerate(message.tx_outs):
...       # если на обрабатываемом выводе имеется такой же
...       # адрес, как заданный адрес, значит, он обнаружен
...       if tx_out.script_pubkey.address(testnet=True) == addr:
...         # вывод UTXO обнаружен; установить поля prev_tx,
...         # prev_index и tx
...         prev_tx = message.hash()
...         prev_index = i
...         prev_amount = tx_out.amount
...         print('found: {}:{}'.format(prev_tx.hex(), prev_index))
found: b2cddd41d18d00910f88c31aa58c6816a190b8fc30f\
e7c665e1cd2ec60efdf3f:7
>>> # создать объект типа TxIn
>>> tx_in = TxIn(prev_tx, prev_index)
```

```
>>> # вычислить сумму на выводе как предыдущую сумму за вычетом
    # платы за транзакцию
>>> output_amount = prev_amount - fee
>>> # создать новый объект типа TxOut для целевого сценария
    # с суммой на выводе транзакции
>>> tx_out = TxOut(output_amount, target_script)
>>> # создать новую транзакцию с одним вводом и одним выводом
>>> tx_obj = Tx(1, [tx_in], [tx_out], 0, testnet=True)
>>> # подписать только ввод данной транзакции
>>> print(tx_obj.sign_input(0, private_key))
True
>>> # сериализировать и хешировать данную транзакцию,
    # чтобы выяснить, как она выглядит
>>> print(tx_obj.serialize().hex())
01000000013fdfef60ecd21c5e667cfe30fcb890a116688ca51ac3880f\
91008dd141ddcdb2070000006b483045022100ff77d2559261df5490ed\
00d231099c4b8ea867e6ccfe8e3e6d077313ed4f1428022033a1db8d69\
eb0dc376f89684d1ed1be75719888090388a16f1e8eedeb8067768012103\
dc585d46cfca73f3a75ba1ef0c5756a21c1924587480700c6eb64e3f75d22
083ffffffff019334e500000000001976a914ad346f8eb57dee9a37981716\
e498120ae80e44f788ac00000000
>>> # отправить подписанную транзакцию через сеть
>>> node.send(tx_obj)
>>> # ожидать секунду, чтобы данное сообщение прошло через сеть,
    # вызвав метод time.sleep(1)
>>> time.sleep(1)
>>> # а теперь запросить данную транзакцию из другого узла сети
>>> # создать объект типа GetDataMessage
>>> getdata = GetDataMessage()
>>> # запросить данную транзакцию, введя ее сообщение
>>> getdata.add_data(TX_DATA_TYPE, tx_obj.hash())
>>> # отправить данное сообщение
>>> node.send(getdata)
>>> # а теперь ожидать ответа на запрос транзакции
>>> received_tx = node.wait_for(Tx)
>>> # если полученная в итоге транзакция обладает таким
    # же самым идентификатором, как и данная транзакция,
    # то реализуемый здесь процесс успешно завершен!
>>> if received_tx.id() == tx_obj.id():
... print('success!')
success!
```

Предметный указатель

М

П

С

T

O'REILLY®

Python для программирования криптовалют

Это руководство поможет вам разобраться в технологии биткойна. Его автор, Джимми Сонг, являющийся одним из ведущих специалистов, обучающих программированию биткойна, поясняет программирующим на языке Python разработчикам, как приступить к построению библиотеки для биткойна "с чистого листа". В книге излагаются основы этой популярной ныне криптовалюты, в том числе математический аппарат, криптографические понятия, блоки и транзакции, а также ее платежная система в виде цепочки боков (или блокчейна).

Прочитав книгу и проработав предлагаемые в ней упражнения, вы сможете уяснить принцип и внутренний механизм действия данной криптовалюты в ходе программирования всех необходимых составляющих библиотеки для биткойна. Из книги вы узнаете, как создавать транзакции, получать данные из одноранговой сети и отсылать транзакции, используя сетевой протокол. Исследуете ли вы приложения биткойна для своей организации или ищете новый путь для развития своей карьеры разработчика, это практическое пособие поможет вам заложить прочный фундамент знаний в области программирования криптовалют.

Узнайте, как:

- **Выполнять синтаксический анализ, проверять на достоверность и создавать биткойновые транзакции**
- **Изучить язык Script, используемый для написания умных контрактов и положенный в основу биткойна**
- **Выполнять упражнения в каждой главе, чтобы построить с нуля библиотеку для биткойна, а также разобраться, каким образом подтверждение работы делает блокчейн безопасным**
- **Программировать биткойн, используя версию Python 3**
- **Понять, каким образом действуют упрощенная проверка оплаты и "тонкие" кошельки**
- **Пользоваться криптографией с открытым ключом и криптографическими примитивами**

"Упражнения, предлагаемые в этой книге, не только представят внутренний механизм работы блокчейна, но и позволят ясно почувствовать все изящество и прелесть данной технологии".

Кен Лю,
лауреат литературных премий Nebula, Hugo и World Fantasy. Его научно-фантастический рассказ о блокчейне *Byzantine Empathy* (Византийская эмпатия) был первоначально опубликован в издательстве MIT Press.

Джимми Сонг — разработчик с более чем 20-летним стажем, в том числе он обладает 5-летним опытом работы с биткойном. Он является редактором интернет-издания `bitcointechtalk.com` и венчурным партнером предприятия Blockchain Capital, также он пишет статьи для *Bitcoin Magazine* и преподает курс программирования биткойна в Университете штата Техас. Он внес свой посильный вклад в самые разные проекты биткойна с открытым кодом, включая Armory, Bitcoin Core, btcd и pycoin

ISBN 978-5-907144-82-8

9 785907 144828 19117

Категория: электронная торговля
Предмет рассмотрения: биткойн
Уровень: промежуточный/продвинутый

ДИАЛЕКТИКА
http://www. williamspublishing.com
http://oreilly.com