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.

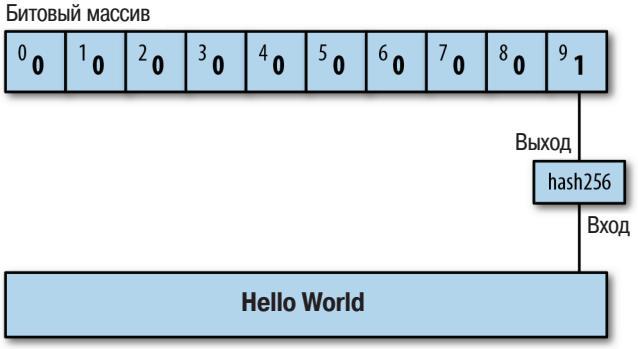

Рис. 12.1. 10-разрядный фильтр Блума с одним элементом

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.

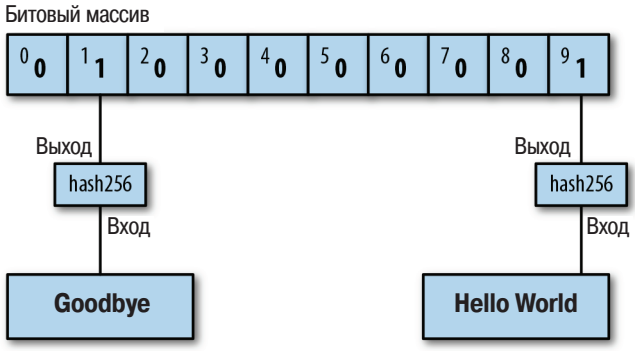

Рис. 12.2. 10-разрядный фильтр Блума с двумя элементами

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.

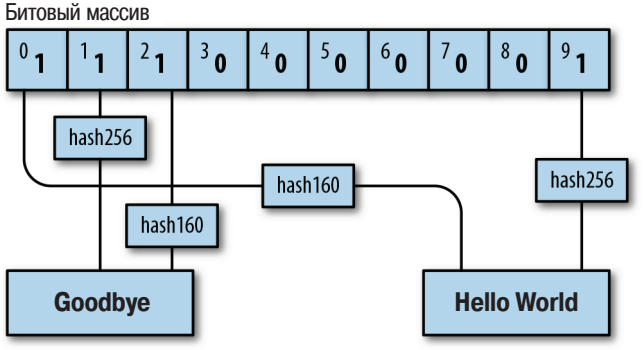

Рис. 12.3. 10-разрядный фильтр Блума с двумя элементами и хеш-функциями

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.

```
0a4000600a080000010940050000006300000000

- 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.

```
02030000030eb2540c41025690160a1014c577061596e32e426b712c7ca000
00000000000030000001049847939585b0652fba793661c361223446b6fc410
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.