

Буквально пару лет назад было немало людей, считавших Bitcoin либо игрушкой для гиков, либо в лучшем случае инструментом для покупки наркотиков, а кое-где его даже хотели запретить. Тем не менее сегодня со всех сторон слышно, что эта технология совсем скоро изменит мир, блокчейн стартапы появляются как грибы после дождя, а криптовалюты, по-моему, уже никто не считает.

Удивительно, но при этом литературы, описывающей принципы работы Bitcoin, до сих пор очень мало, а та что есть, как правило, несет никакой ценной информации. Поэтому я написал небольшую книгу для тех, кто хочет понять, как **действительно** работает Bitcoin. В ней я постарался коснуться всех основных моментов и по возможности объяснил их максимально доступным языком. Хотя книга рассчитана все-таки на людей, близких к IT сфере, благо примеров кода, специфической терминологии и несмешных шуток хватает.



"You can't stop things like Bitcoin. It will be everywhere and the world will have to readjust. World governments will have to readjust" - John McAfee, Founder of McAfee

Дисклеймер

Я подразумеваю, что ты, уважаемый читатель, хотя бы в общих чертах понимаешь, что такое Bitcoin. То есть ты по крайней мере слышал такие слова, как [blocks](#), [transactions](#), [private / public keys](#), [mining](#). Совсем хорошо, если уже встречался с терминами вроде [ECDSA](#), [Proof-of-Work](#) или даже [51% attack](#).

В случае, если большая часть этих выражений для тебя в новинку, рекомендую сначала потратить пару часов и посмотреть [раз](#), [два](#), [три](#). Ну или полистать

- [Что такое биткоин?](#) by Coinspot
- [Как работает Биткойн?](#) by Bitcoin.org
- [Bitcoin](#) by Lurkmore
- [Bitcoin. Как это работает](#) by Habrahabr

Оглавление

1. [Bitcoin in a nutshell - Cryptography !\[\]\(467d80e979964f7f8c752fb22248b5b7_img.jpg\)](#)
2. [Bitcoin in a nutshell - Transaction !\[\]\(b71552d33dbf62adf5e5199a70ee02bf_img.jpg\)](#)
3. [Bitcoin in a nutshell - Protocol !\[\]\(03134b765d1473836ff001925b1b0550_img.jpg\)](#)
4. [Bitcoin in a nutshell - Blockchain !\[\]\(aed6947356668967079310026052edc0_img.jpg\)](#) // Soon
5. [Bitcoin in a nutshell - Mining !\[\]\(e61aeb0d9066d5d9e54d9b655f50da3d_img.jpg\)](#) // Soon

Ссылки для общего развития

1. ["Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto](#)
2. ["Mastering Bitcoin" By Andreas M. Antonopoulos](#)
3. ["Blockchain" By Melanie Swan](#)
4. [Bitcoin developer documentation](#)
5. [Bitcoin developer guide](#)
6. [/r/Bitcoin](#)
7. [bitcoin.stackexchange.com](#)
8. [Bitcoin wiki](#)
9. ["Mastering Bitcoin" online](#)

Проект распространяется под лицензией MIT. Исходники в markdown плюс фрагменты кода [здесь](#), contributions welcome.

Одна из причин, почему Bitcoin продолжает привлекать столько внимания - это его исключительная "математичность". Сатоши Накамото удалось создать систему, которая способна функционировать при полном отсутствии доверия между ее участниками. Все взаимодействия основаны на строгой математике, никакого человеческого фактора - вот в чем была революционность идеи, а не в одноранговой сети, как многие думают. Поэтому первую главу я решил посвятить именно математическим основам Bitcoin.

Ниже я постараюсь объяснить вам самые базовые вещи - эллиптические кривые, ECC, приватные / публичные ключи и так далее. По возможности я буду иллюстрировать свои слова на Python, если что-то непонятно - спрашивайте в комментариях.



1. Introduction
2. Elliptic curve
3. Digital signature
4. Private key
5. Public key
6. Formats & address
7. Sign
8. Verify
9. Formats
10. Links

Introduction

Как я уже сказал выше, криптография - это фундаментальная часть Bitcoin. Без нее вообще бы ничего не заработало, поэтому начинать нужно именно отсюда.

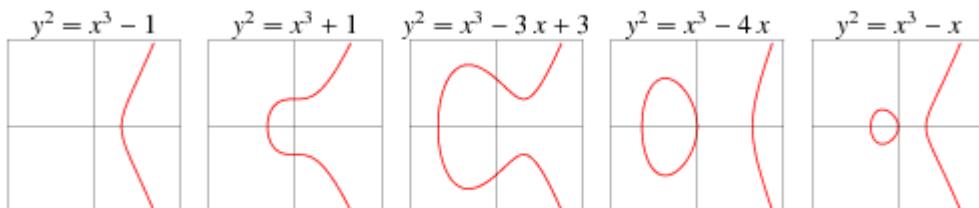
В Bitcoin используется так называемая [криптография на эллиптических кривых](#) (*Elliptic curve cryptography, ECC*). Она основана на некоторой особой функции - эллиптической кривой (не путать с эллипсом). Что это за функция и чем она так примечательна я расскажу дальше.

Elliptic curve

Эллиптическая кривая над полем K — неособая кубическая кривая на проективной плоскости над \hat{K} (алгебраическим замыканием поля K), задаваемая уравнением 3-й степени с коэффициентами из поля K и «точкой на бесконечности» - [Wikipedia](#)

Если на пальцах, то эллиптическая кривая - это внешне довольно простая функция, как правило, записываемая в виде так называемой формы Вейерштрасса: $y^2 = x^3 + ax + b$

В зависимости от значений параметров a и b , график данной функции может выглядеть по разному:



Скрипт для отрисовки графика на Python:

```

import numpy as np
import matplotlib.pyplot as plt

def main():
    a = -1
    b = 1

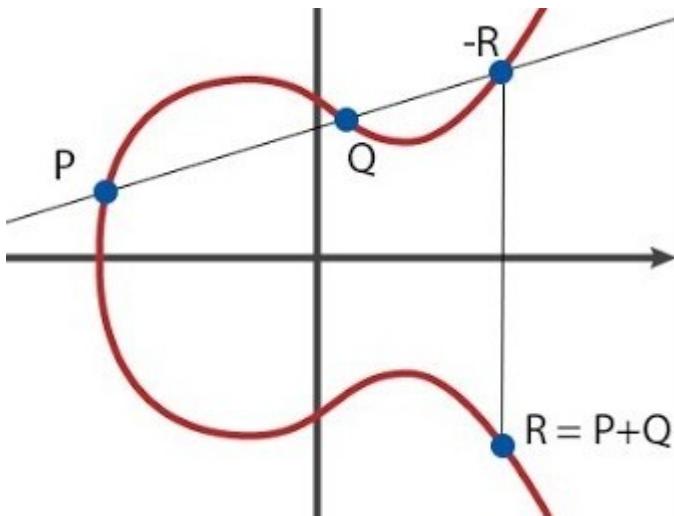
    y, x = np.ogrid[-5:5:100j, -5:5:100j]
    plt.contour(x.ravel(), y.ravel(), pow(y, 2) - pow(x, 3) - x * a - b, [0])
    plt.grid()
    plt.show()

if __name__ == '__main__':
    main()

```

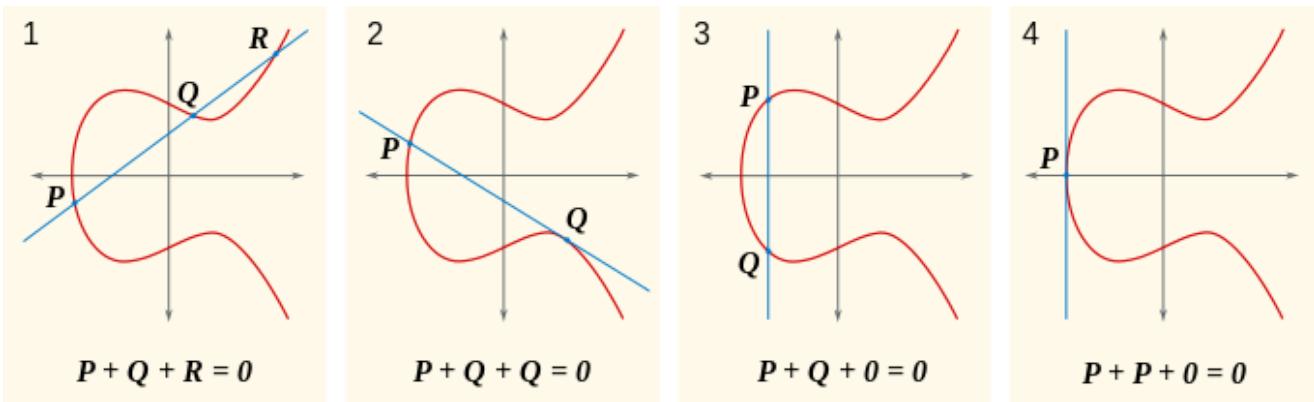
Если верить вики, то впервые эта функция засветилась еще в трудах Диофанта, а позже, в 17 веке, ей заинтересовался сам Ньютон. Его исследования во многом привели к формулам сложения точек на эллиптической кривой, с которыми мы сейчас познакомимся. Здесь и в дальнейшем мы будем рассматривать некоторую эллиптическую кривую α .

Пусть есть две точки $P, Q \in \alpha$. Их суммой называется точка $R \in \alpha$, которая в простейшем случае определяется следующим образом: проведем прямую через P и Q - она пересечет кривую α в единственной точке, назовем ее $-R$. Поменяв y координату точки $-R$ на противоположную по знаку, мы получим точку R , которую и будем называть суммой P и Q , то есть $P + Q = R$.



Считаю необходимым отметить, что мы именно **вводим** такую операцию сложения - если вы будете складывать точки в привычном понимании, то есть складывая соответствующие координаты, то получите совсем другую точку $R'(x_1 + x_2, y_1 + y_2)$, которая, скорее всего, не имеет ничего общего с R или $-R$ и вообще не лежит на кривой α .

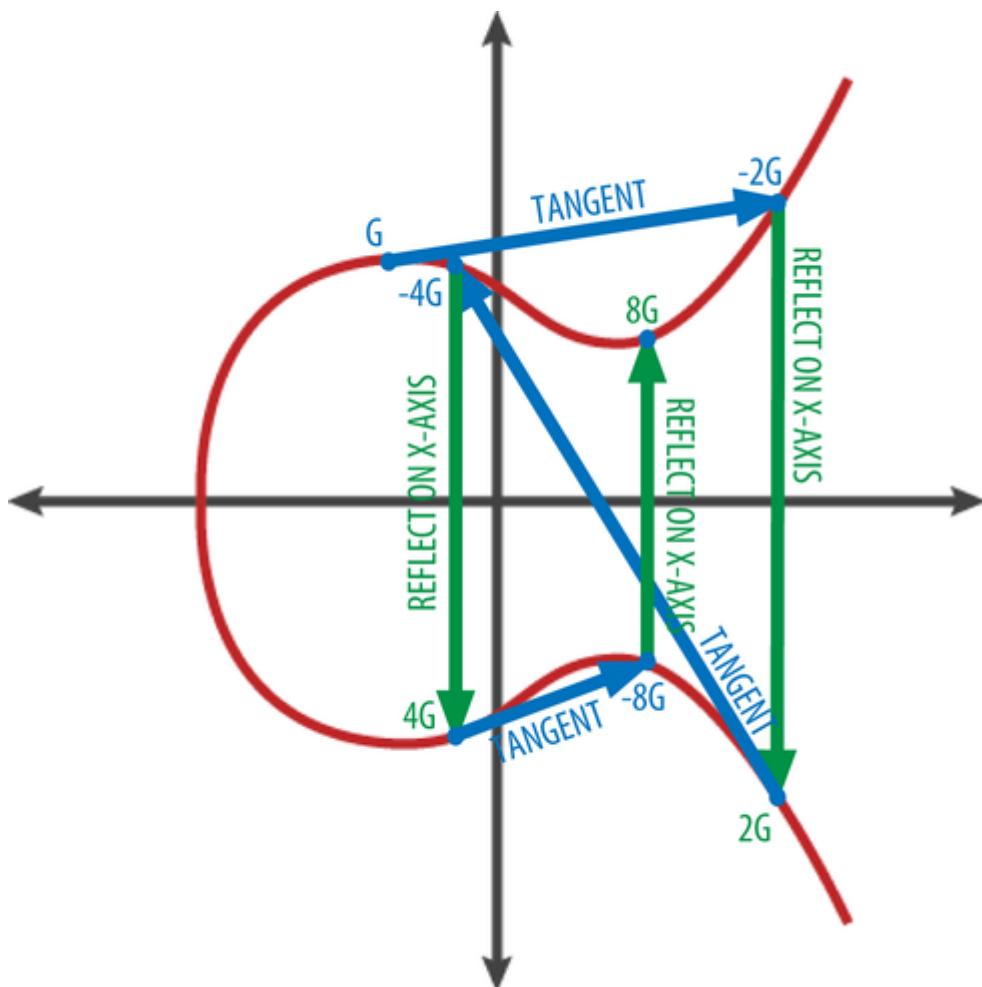
Самые сообразительные уже задались вопросом - а что будет, если например провести прямую через две точки, имеющие координаты вида $P(a, b)$ и $Q(a, -b)$, то есть прямая, проходящая через них, будет параллельна оси ординат (третий кадр на картинке ниже).



Несложно увидеть, что в этом случае отсутствует третье пересечение с кривой α , которое мы называли $-R$. Для того, чтобы избежать этого казуса, введем так называемую **точку в бесконечности** (point of infinity), обозначаемую обычно O или просто 0 , как на картинке. И будем говорить, что в случае отсутствия пересечения $P + Q = O$.

Особый интерес для нас представляет случай, когда мы хотим сложить точку саму с собой (2 кадр, точка Q). В этом случае просто проведем касательную к точке Q и отразим полученную точку пересечения относительно y .

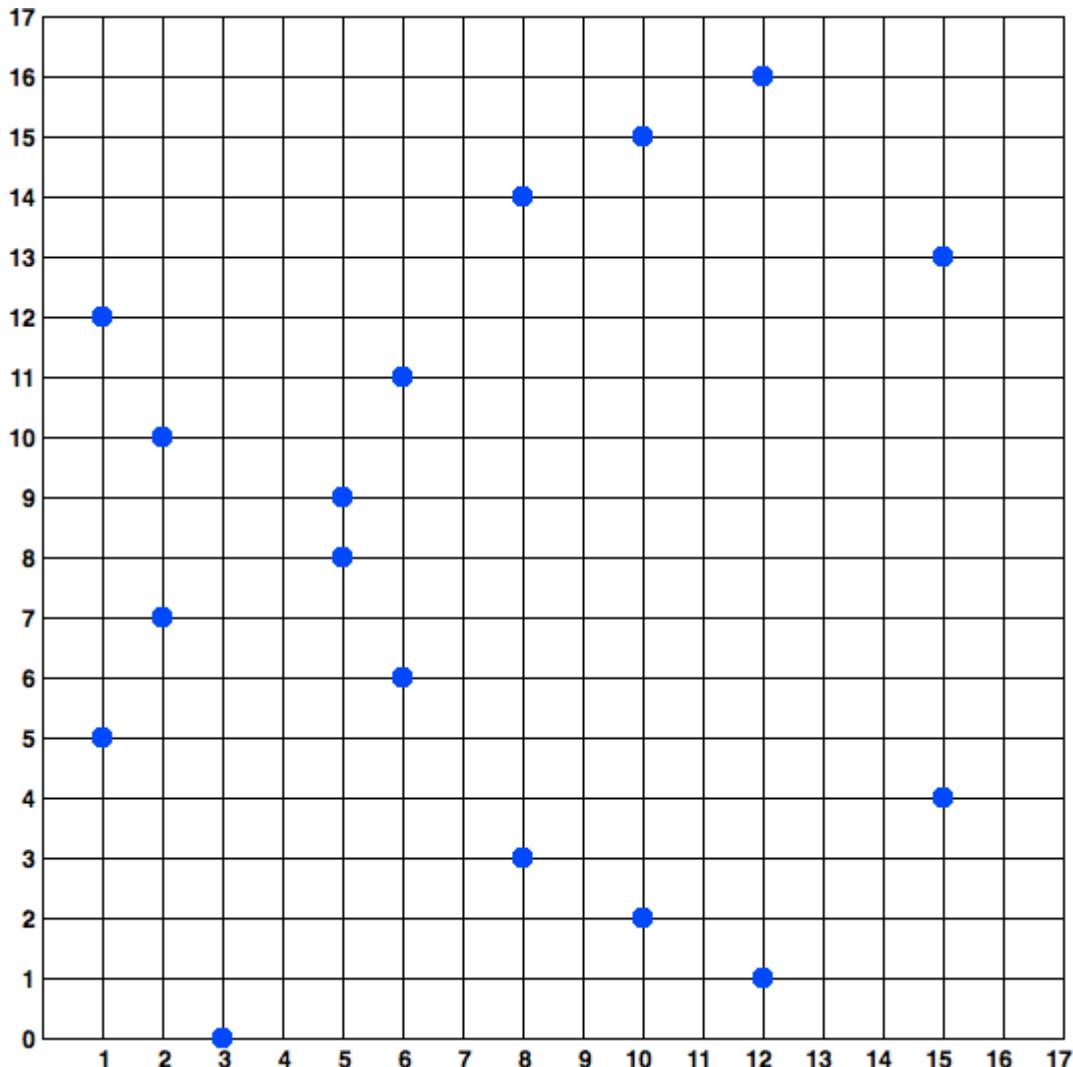
Теперь, легким движением руки, можно ввести операцию умножения точки на какое-то \mathbb{N} число. В результате получим новую точку $K = G * k$, то есть $K = G + G + \dots + G$, k раз. С картинкой все должно стать вообще понятно:



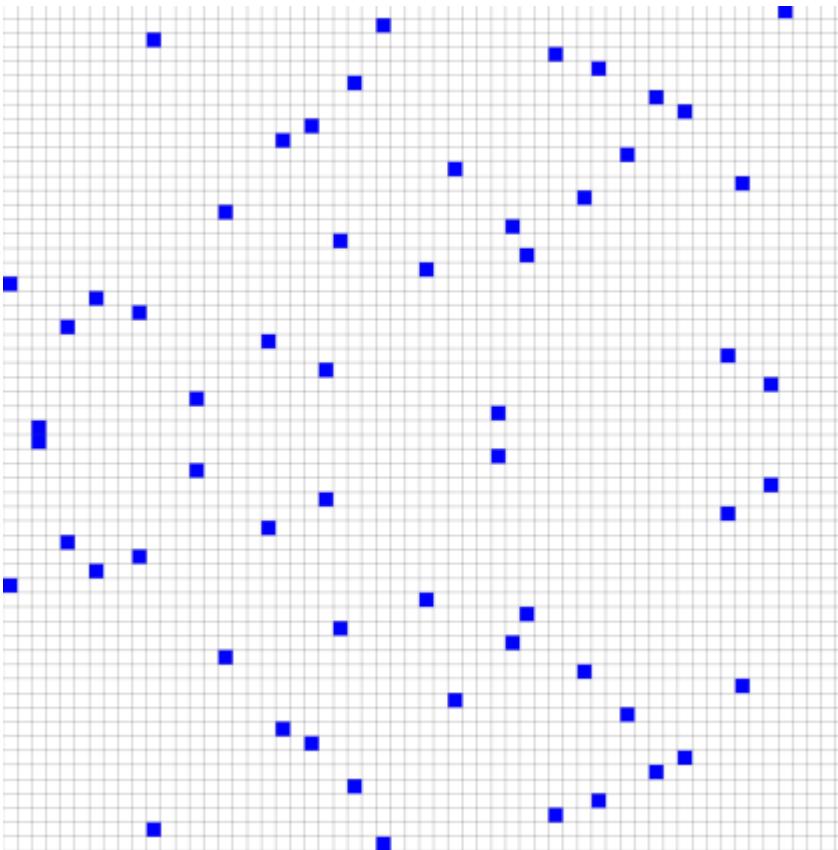
Elliptic curve over a finite field

В ECC используется точно такая же кривая, только рассматриваемая над некоторым конечным полем $F_p = \mathbb{Z}/\mathbb{Z}_p = \{0, 1, \dots, p - 1\}$, где p - простое число. То есть функция приобретает вид $y^2 \bmod p = x^3 + ax + b \pmod p$.

Все названные свойства (сложение, умножение, точка в бесконечности) для такой функции остаются в силе, хотя, если попробовать нарисовать данную функцию, то напоминать привычную эллиптическую кривую она будет лишь отдаленно (в лучшем случае). А понятие "касательной к функции в точке" вообще теряет всякий смысл, но это ничего страшного. Вот пример функции $y^2 = x^3 + 7$ для $p = 17$:



А вот для $p = 59$, тут вообще почти хаотичный набор точек. Единственное, что все еще напоминает о происхождении этого графика, так это симметрия относительно оси X .



P. S. Если вам интересно, как в случае с кривой над конечным полем вычислить координаты точки $R(x_3, y_3)$, зная координаты $P(x_1, y_1)$ и $Q(x_2, y_2)$ - можете полистать "[An Introduction to Bitcoin, Elliptic Curves and the Mathematics of ECDSA](#)" by N. Mistry, там все подробно расписано, достаточно знать математику на школьном уровне.

P. P. S. На случай, если мои примеры не удовлетворили ваш пытливый ум, вот [сайт](#) для рисования кривых всех сортов, поэкспериментируйте.

SECP256k1

Возвращаясь к Bitcoin, в нем используется кривая [SECP256k1](#). Она имеет вид $y^2 = x^3 + 7$ и рассматривается над полем F_p , где p - очень большое простое число, а именно $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

Так же для SECP256k1 определена так называемая *base point*, она же *generator point* - это просто точка, как правило, обозначаемая G , лежащая на данной кривой. Она нужна для создания публичного ключа, о котором будет рассказано ниже.

Простой пример: используя Python, проверим, принадлежит ли точка $G(x, y)$ кривой SECP256k1

```
>>> p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7) % p == y**2 % p
True
```

Digital signature

Электронная подпись (ЭП), Электронная цифровая подпись (ЭЦП) — реквизит электронного документа, полученный в результате криптографического преобразования информации с использованием закрытого ключа подписи и позволяющий проверить отсутствие искажения информации в электронном документе с момента формирования подписи (целостность), принадлежность подписи владельцу сертификата ключа подписи (авторство), а в случае успешной проверки подтвердить факт подписания электронного документа (неотказуемость) - [Wikipedia](#)

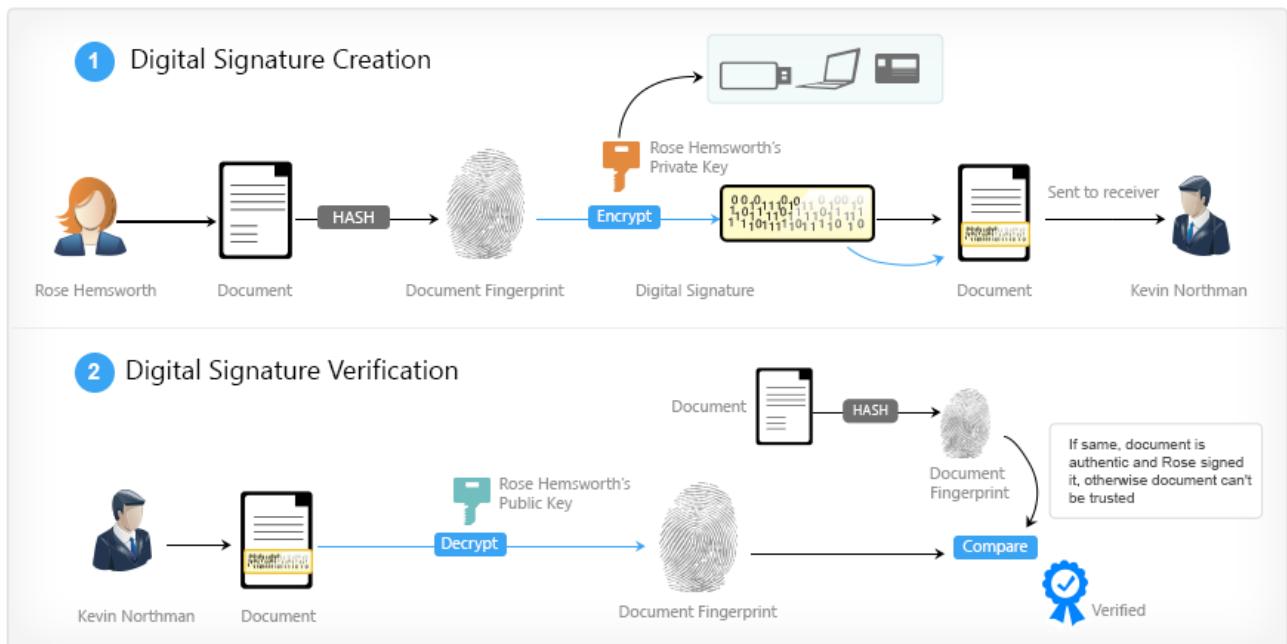
Общая идея такая: Алиса хочет перевести 1 BTC Бобу. Для этого она создает сообщение типа:

```
{  
  "from" : "1FXySbm7jpJfHEJRjSNPPUqnpRTcSuS8aN", // Alice's address  
  "to" : "1Eqm3z1yu6D4Y1c1LXKqReqo1gvZNrmfvN", // Bob's address  
  "amount" : 1 // Send 1 BTC  
}
```

Потом Алиса берет свой приватный ключ (пока что можете считать, что это число, известное только Алисе), хэш сообщения и функцию вида `sign_text(private_key, text)`. На выходе она получает подпись своего сообщения - в случае ECDSA это будет пара целых чисел, для других алгоритмов подпись может выглядеть по другому. После этого она рассыпает всем участникам сети исходное сообщение, подпись и свой публичный ключ.

В результате, каждый Вася при желании сможет взять эту троицу, функцию вида

`validate_signature(public_key, signature, text)` и проверить, действительно ли владелец приватного ключа подписывал это сообщение или нет. А если внутри сети все знают, что `public\key` принадлежит Алисе, то можно понять, отправила эти деньги она или же кто-то пытается сделать это от ее имени.



Более того, предположим, что нашелся человек, вставший между Алисой и остальной сетью. Пусть он перехватил сообщение Алисы и что-то в нем изменил, буквально 1 бит из миллиарда. Но даже в этом случае проверка подписи на валидность `validate_signature(public_key, signature, text')` покажет, что сообщение было изменено.

Это очень важная фича для Bitcoin, потому как сеть *распределенная*. Вы не можете знать заранее, к кому попадет ваша транзакция с требованием перевести 1000 BTC. Но изменить ее (например указать свой адрес с качестве получателя) никто не сможет, потому как транзакция подписана вашим приватным ключом, и остальные участники сети сразу поймут, что здесь что-то не так.

AHTUNG! В действительности процесс довольно сильно отличается от вышеописанного. Здесь я просто на пальцах показал, что из себя представляет электронно-цифровая подпись и зачем она нужна. Реальный алгоритм описан в главе "[Bitcoin in a nutshell - Transactions](#)".

Private key

Приватный ключ - это довольно общий термин и в различных алгоритмах электронной подписи могут использоваться различные типы приватных ключей.

Как вы уже могли заметить, в Bitcoin используется алгоритм ECDSA - в его случае приватный ключ - это некоторое натуральное 256 битное число, то есть самое обычное целое число от 0 до $2^{256} - 1$. Технически, даже число 123456 будет являться корректным приватным ключом, но очень скоро вы узнаете, что ваши монеты "принадлежат" вам ровно до того момента, как у злоумышленника окажется ваш приватный ключ, а значения типа 123456 очень легко перебираются.

Важно отметить, на сегодняшний день перебрать все ключи невозможно в силу того, что 2^{256} - это фантастически большое число.

Постараемся его представить: согласно [этой статье](#), на всей Земле немногим меньше 10^{22} песчинок. Воспользуемся тем, что $10^3 \approx 2^{10}$, то есть $10^{22} \approx 2^{80}$ песчинок. А всего адресов у нас 2^{256} , примерно 2^{80^3} .

Значит, мы можем взять весь песок на Земле, превратить каждую песчинку в новую Землю, в получившейся куче планет каждую песчинку на каждой планете снова превратить в новую Землю, и суммарное число песчинок все равно будет на порядки меньше числа возможных приватных ключей.

По этой же причине большинство Bitcoin клиентов при создании приватного ключа просто берут 256 случайных бит - вероятность коллизии крайне мала.

Python

```
>>> import random
>>> private_key = ''.join(['%x' % random.randrange(16) for x in range(0, 64)])
>>> private_key
'9ceb87fc34ec40408fd8ab3fa81a93f7b4ebd40bba7811ebef7cbc80252a9815'
>>> # or
>>> import os
>>> private_key = os.urandom(32).encode('hex')
>>> private_key
'0a56184c7a383d8bcce0c78e6e7a4b4b161b2f80a126caa48bde823a4625521f'
```

Python, [ECDSA](#)

```
>>> import binascii
>>> import ecdsa # sudo pip install ecdsa
>>> private_key = ecdsa.SigningKey.generate(curve=ecdsa.SECP256k1)
>>> binascii.hexlify(private_key.to_string()).decode('ascii').upper()
u'CE47C04A097522D33B4B003B25DD7E8D7945EA52FA8931FD9AA55B315A39DC62'
```

Bitcoin-cli

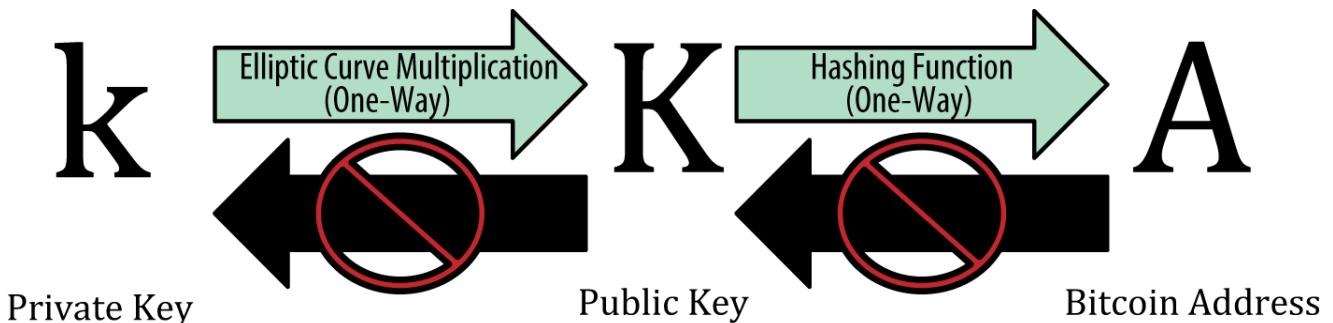
```
$ bitcoin-cli getnewaddress
14RVpC4su4PzSafjCKVWP2YBHv3f6zNf6U
$ bitcoin-cli dumpprivkey 14RVpC4su4PzSafjCKVWP2YBHv3f6zNf6U
L3SPdkFWMnFyDGyV3vkCjroGi4zfD59Wsc5CHdB1LirjN6s2vii9 // Format is different, explanation below
```

Public key

Пусть k - наш приватный ключ, G - base point, тогда публичный ключ $K = G * k$. То есть, фактически, публичный ключ - это некоторая точка, лежащая на кривой SECP256k1.

Два важных нюанса. Во-первых, несложно видеть, что операция получения публичного ключа определена однозначно, то есть конкретному приватному ключу всегда соответствует один единственный публичный ключ. Во-вторых, обратная операция является вычислительно трудной и, в общем случае, получить приватный ключ из публичного можно только полным перебором первого.

Ниже вы узнаете, что точно такая же связь существует между публичным ключом и адресом, только там все дело в необратимости хэш-функций.



Python, ECDSA

```
>>> import binascii
>>> import ecdsa
>>> private_key = ecdsa.SigningKey.generate(curve=ecdsa.SECP256k1)
>>> public_key = private_key.get_verifying_key()
>>> binascii.hexlify(public_key.to_string()).decode('ascii').upper()
u'D5C08F1BFC9C26A5D18FE9254E7923DEBB34AFB92AC23ABFC6388D2659446C1F04CCDEBB677EAABFED9294663EE79D
71B57CA6A6B76BC47E6F8670FE759D746'
```

C++, [libbitcoin](#)

```
#include <bitcoin/bitcoin.hpp>
#include <iostream>

int main() {
    // Private key
    bc::ec_secret secret = bc::decode_hash(
        "038109007313a5807b2ecc082c8c3fb988a973cacf1a7df9ce725c31b14776");
    // Get public key
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;
}
```

Для компиляции и запуска используем (предварительно установив [libbitcoin](#)):

```
$ g++ -o public_key <filename> $(pkg-config --cflags --libs libbitcoin)
$ ./public_key
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
```

Вы можете видеть, что форматы публичных ключей в первом и втором примере отличаются (как минимум длиной), об этом я подробнее расскажу ниже.

Formats & address

Base58Check encoding

Эта кодировка была придумана специально для Bitcoin, поэтому стоит понимать, как она работает и зачем она вообще нужна. Ее суть в том, чтобы максимально кратко записать последовательность байт в удобочитаемом формате и при этом свести вероятность возможных опечаток к минимуму. Я думаю, вы сами понимаете, что в случае Bitcoin безопасность лишней не бывает. Один неправильный символ и деньги уйдут на адрес, ключей к которому скорее всего никто никогда не найдет. Вот комментарий к кодировке из в [base58.h](#):

```
// Why base-58 instead of standard base-64 encoding?
// - Don't want 0011 characters that look the same in some fonts and
//   could be used to create visually identical looking account numbers.
// - A string with non-alphanumeric characters is not as easily accepted as an account
//   number.
// - E-mail usually won't line-break if there's no punctuation to break at.
// - Doubleclicking selects the whole number as one word if it's all alphanumeric.
```

Краткость записи проще всего реализовать, используя довольно распространенную кодировку [Base64](#), то есть используя систему счисления с основанием 64, где для записи используются цифры `0,1,...,9`, буквы `a-z` и `A-Z` - это дает 62 символа, оставшиеся два могут быть чем угодно, в зависимости от реализации.

Первое отличие [Base58Check](#) в том, что убраны символы `0,0,1,I` на случай, если кто-нибудь решит их перепутать. Получается 58 символов, можете проверить

```

b58 = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

def base58encode(n):
    result = ''
    while n > 0:
        result = b58[n % 58] + result
        n /= 58
    return result

# print "Base58 encode for '123123':", base58encode(123123)
# # Base58 encode for '123123': dbp

```

Второе отличие - это тот самый *check*. В конец строки добавляется *checksum* - первые 4 байта `SHA256(SHA256(str))`. И еще нужно записать в начало столько единиц, сколько ведущих нулей было до кодировки в base58, это уже дело техники.

```

import hashlib

def base58encode(n):
    b58 = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    result = ''
    while n > 0:
        result = b58[n % 58] + result
        n /= 58
    return result

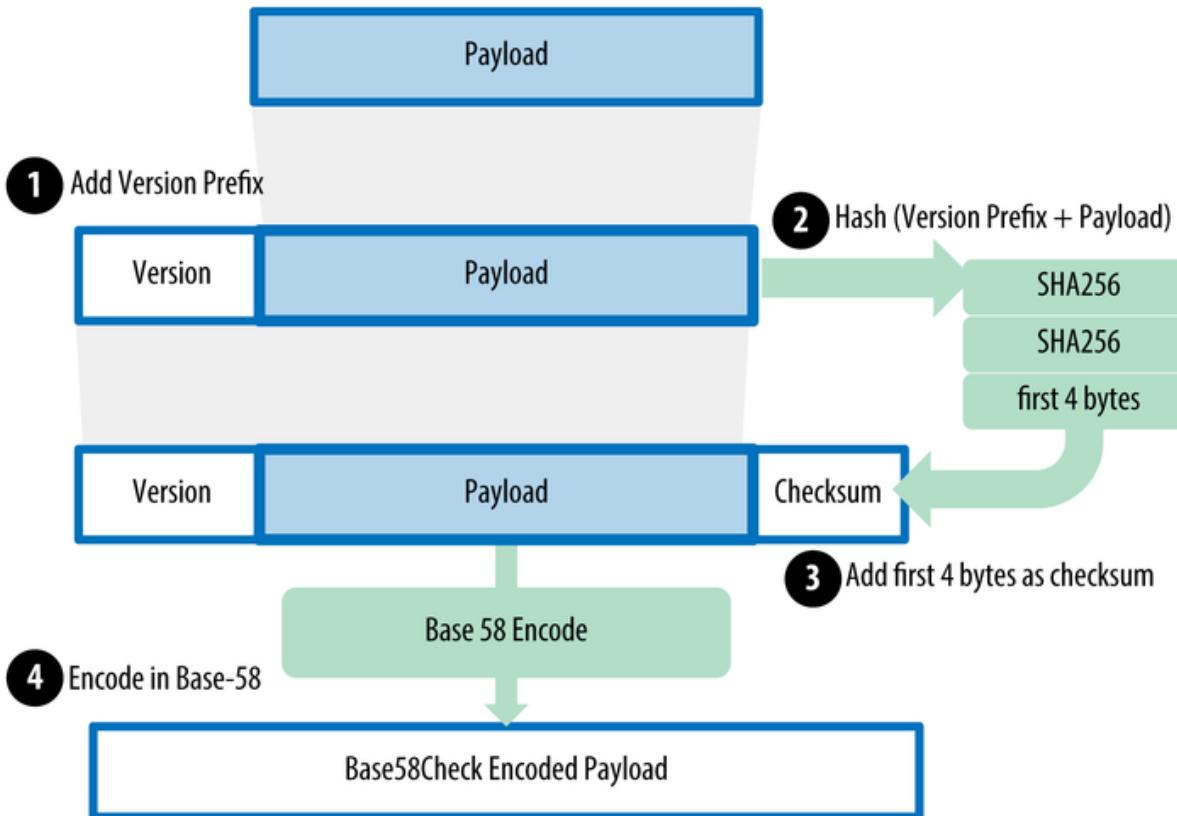
# Will be used to decode raw bytes
def base256decode(s):
    result = 0
    for c in s:
        result = result * 256 + ord(c)
    return result

def countLeadingZeroes(s):
    count = 0
    for c in s:
        if c == '\0':
            count += 1
        else:
            break
    return count

def base58CheckEncode(prefix, payload):
    s = chr(prefix) + payload
    checksum = hashlib.sha256(hashlib.sha256(s).digest()).digest()[0:4]
    result = s + checksum
    return '1' * countLeadingZeroes(result) + base58encode(base256decode(result))

```

Base58Check Encoding



Private key formats

Самый очевидный способ хранить приватный ключ - это записать 256 бит в виде кучи нулей и единиц. Но, наверное, любой технически грамотный человек понимает, что будет сильно проще представить ту же самую последовательность в виде 32 байт, где каждому байту соответствует два символа в шестнадцатиричной записи. Напомню, что в этом случае используются цифры `0, 1, ..., 9` и буквы `A, B, C, D, E, F`. Этот формат я использовал в примерах выше, для красоты его еще иногда разделяют пробелами.

```
E9 87 3D 79 C6 D8 7D C0 FB 6A 57 78 63 33 89 F4 45 32 13 30 3D A6 1F 20 BD 67 FC 23 3A A3 32 62 ↴
```

Другой, более прогрессивный формат - [WIF](#) (*Wallet Import Format*). Строится он довольно просто:

1. Берем приватный ключ, например

```
0C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D
```

2. Записываем его в Base58Check с префиксом `0x80`. Все.

```

private_key = '0a56184c7a383d8bcce0c78e6e7a4b4b161b2f80a126caa48bde823a4625521f'

def privateKeyToWif(key_hex):
    return base58CheckEncode(0x80, key_hex.decode('hex'))

# print "Private key in WIF format:", privateKeyToWif(private_key)
# # Private key in WIF format: 5HtqcFguVHA22E3bcjJR2p4HHMEGnEXxVL5hnxmPQvRedSQSuT4

# OR

from pybitcoin import BitcoinPrivateKey
private_key =
BitcoinPrivateKey('0a56184c7a383d8bcce0c78e6e7a4b4b161b2f80a126caa48bde823a4625521f')
# print "Private key in WIF format:", private_key.to_wif()
# # Private key in WIF format: 5HtqcFguVHA22E3bcjJR2p4HHMEGnEXxVL5hnxmPQvRedSQSuT4

```

Public key formats

На всякий случай напомню, что публичный ключ - это просто точка на прямой SECP256k1. Первый и самый распространенный вариант его записи - *uncompressed* формат, по 32 байта для X и Y координат. В этом случае используется префикс `0x04` и того 65 байт.

```

import ecdsa

private_key = '0a56184c7a383d8bcce0c78e6e7a4b4b161b2f80a126caa48bde823a4625521f'

def privateKeyToPublicKey(s):
    sk = ecdsa.SigningKey.from_string(s.decode('hex'), curve=ecdsa.SECP256k1)
    vk = sk.verifying_key
    return ('\x04' + sk.verifying_key.to_string()).encode('hex')

uncompressed_public_key = privateKeyToPublicKey(private_key)

# print "Uncompressed public key: {}, size: {}".format(uncompressed_public_key,
len(uncompressed_public_key) / 2)
# # Uncompressed public key:
045fbbe96332b2fc2bcc1b6a267678785401ee3b75674e061ca3616bbb66777b4f946bdd2a6a8ce419eacc5d05718bd71
8dc8d90c497cee74f5994681af0a1f842, size: 65

```

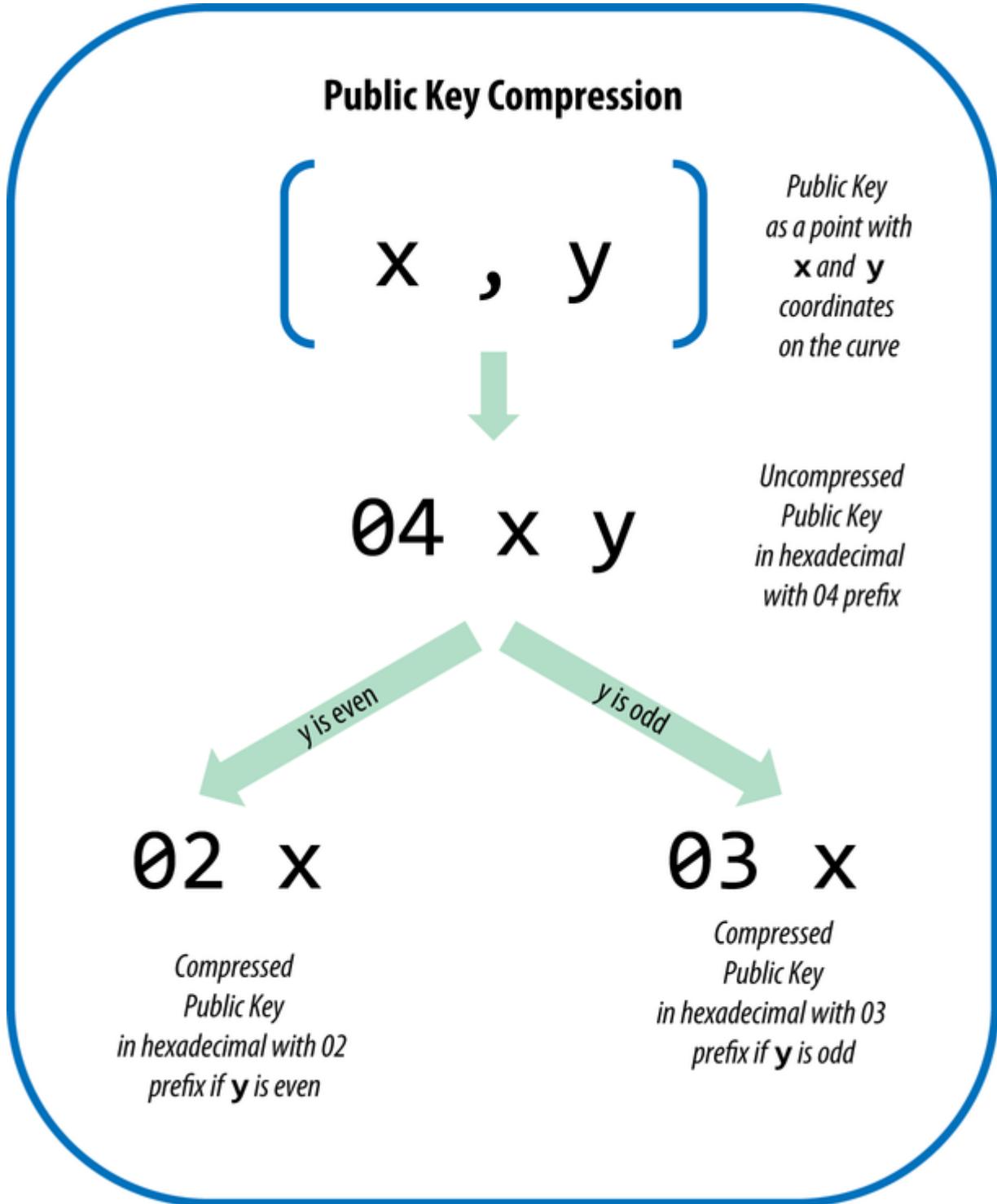
Однако, как можно догадаться из названия, это не самый оптимальный способ хранить публичный ключ.

Вы удивитесь, но второй формат называется *compressed*. Суть его в следующем: публичный ключ - это точка на кривой, то есть пара чисел удовлетворяющая уравнению $y^2 \bmod p = x^2 + ax + b \bmod p$. А значит можно записать только X координату и если нам понадобится Y координата - просто решаем уравнение. Тем самым мы уменьшаем размер публичного ключа почти на 50% !

Единственный нюанс - если точка лежит на кривой, то для ее X координаты очевидно существует два решения такого уравнения (посмотрите на графики выше, если сомневаетесь). Обычно мы бы просто сохранили знак для Y координаты, но когда речь идет о функции над конечным полем, то нужно воспользоваться следующим свойством: **если для X координаты существуют решения уравнения,**

то одна из точек будет иметь четную Y координату, а вторая - нечетную (опять же, можете сами в этом убедиться).

В первом случае используется префикс `0x02`, во втором - `0x03`. Вот иллюстрация процесса:



Address

Как уже было сказано, адрес получается из публичного ключа однозначным образом. Более того, провести обратную операцию невозможно, так как используются криптографически стойкие хэш функции - [RIPEMD160](#) и [SHA256](#). Вот алгоритм перевода публичного ключа в адрес:

1. Возьмем приватный ключ, например

45b0c38fa54766354cf3409d38b873255dfa9ed3407a542ba48eb9cab9dfca67

2. Получим из него публичный ключ в *uncompressed* формате, в данном случае это

04162ebcd38c90b56fdbdb4b0390695afb471c944a6003cb334bbf030a89c42b584f089012beb4842483692bdff9fcab8
676fed42c47bffb081001209079bbc8db .

3. Считаем `RIPEMD160(SHA256(public_key))`, получается 5879DB1D96FC29B2A6BDC593E67EDD2C5876F64C

4. Переводим результат в *Base58Check* с префиксом `0x00` - 17JdJpDyu3tB5GD3jwZP784W5KbRdfb84X . Это и есть адрес.

```
def pubKeyToAddr(s):
    ripemd160 = hashlib.new('ripemd160')
    ripemd160.update(hashlib.sha256(s.decode('hex')).digest())
    return base58CheckEncode(0, ripemd160.digest())

def keyToAddr(s):
    return pubKeyToAddr(privateKeyToPublicKey(s))

# print keyToAddr("45b0c38fa54766354cf3409d38b873255dfa9ed3407a542ba48eb9cab9dfca67")
# # '17JdJpDyu3tB5GD3jwZP784W5KbRdfb84X'
```

Sign & verify

Не думаю, что вам нужно обязательно знать технические подробности того, как именно *ECDSA* подписывает и проверяет сообщения, все равно здесь вы точно будете пользоваться готовыми библиотеками. Главное, чтобы у вас было общее понимание того, зачем это нужно, но если вам все таки интересно - полистайте [Layman's Guide to Elliptic Curve Digital Signatures](#), там внизу есть красивая визуализация всего процесса, можете сами попробовать.

У меня на этом все, следующая глава: [Bitcoin in a nutshell - Transaction](#).

Links

- ["Mastering Bitcoin" - Keys, Addresses, Wallets](#)
- ["An Introduction to Bitcoin, Elliptic Curves and the Mathematics of ECDSA" by N. Mistry](#)
- ["Secure Implementation of ECDSA Signatures in Bitcoin" by Di Wang](#)
- [Elliptic Curve Cryptography: a gentle introduction](#)
- [Эллиптическая криптография: теория](#)
- [Generating A Bitcoin Private Key And Address](#)
- [Generating EC keypair, signing and verifying ECDSA signature](#)

Если говорить об уже существующей банковской системе, то транзакция внутри какого-нибудь Альфа-банка - это просто редактирование таблицы балансов, где уменьшается число напротив одного имени и увеличивается напротив другого. В случае с межбанковскими переводами подключаются некоторые сторонние организации, например SWIFT, но, по сути, все работает примерно так же.

Когда мы имеем дело с финансовой системой на основе блокчейна, то процесс денежного перевода выглядит совершенно иначе. В Bitcoin не существует никакой общей таблицы вида <адрес, баланс>, ровно как и не существует регулятора, который бы эту таблицу редактировал. В этой статье я покажу, что из себя представляет транзакция в Bitcoin, как она строится, и объясню, зачем же внутри Bitcoin добавлен свой язык программирования, про который все слышали, но никто не видел.



Table of content

1. Introduction
2. Inputs & outputs
3. Fee
4. UTXO
5. Txn structure
6. Script
7. Lock & unlock transaction
8. Password script
9. Pay to Public Key Hash (P2PKH)
10. P2P storage
11. Links

Introduction

Как я уже сказал выше, в Bitcoin не существует никакой единой структуры, в которой каждому адресу был бы сопоставлен его текущий баланс. Вместо этого используется тот самый пресловутый блокчейн, то есть хранятся вообще все транзакции. Для простоты пока что можете считать, что это сообщения вида

```
<address 1> sent <amount> BTC to <address 2>
```

А значит, если пройтись по всему блокчейну, то можно посчитать, сколько монет "принадлежит" конкретному адресу.

Inputs & outputs

Реальная транзакция в сети Bitcoin, на самом деле, немного сложнее описанной выше. В действительности, это некоторая громоздкая структура, главными составляющими которой являются входы (inputs) и выходы (outputs).

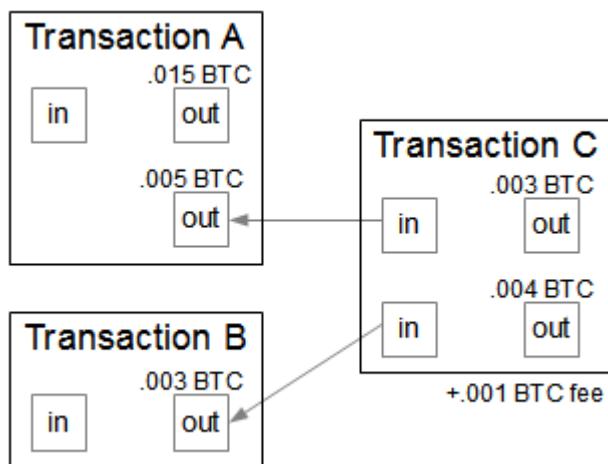
Inputs - это транзакции, на которые вы "ссылаетесь". Представим, что на ваш адрес **X** когда-то было отправлено три транзакции:

- **TXN_ID** - 123456, **VALUE** - 40 BTC
- **TXN_ID** - 6453795, **VALUE** - 10 BTC
- **TXN_ID** - 888888, **VALUE** - 100 BTC

Если вам нужно потратить, например, 45 BTC, то вы можете сослаться на транзакцию **888888**, или сразу на две транзакции: **123456** и **6453795**. При желании вы можете даже сослаться на все три транзакции, правда непонятно зачем.

Outputs - дословно "выходы". Пока что можете считать, что это адреса (хотя это не так), на которые в результате исполнения транзакции будут "отправлены" средства. Выходов также может быть несколько, и каждому из них указывается своя сумма.

На картинке ниже создается новая транзакция **C**, которая ссылается на две входящие - **A** и **B**. В результате на входе у транзакции получается **0.008 BTC**, которые потом разделяются на два выхода - на первый адрес отправляется **0.003 BTC**, а на второй **0.004 BTC**.



Возможность указать сразу несколько выходов - это очень важная фича, потому что транзакцию (а если точнее - ее выход) можно использовать как вход только один раз и только целиком. То есть если у вас есть входящая транзакция на 10 BTC, а вам нужно потратить 8 из них в каком-нибудь Старбаксе, вы просто создаете транзакцию с одним входом и двумя выходами: на 8 BTC в магазин и на 2 BTC обратно на свой адрес. Если же вы создадите транзакцию, в которой сумма выходов меньше суммы входов (как на картинке), то разница отправляется на адрес майнера, записавшего вашу транзакцию в блок.

Fee

Именно эта разница между суммой входов и суммой выходов называется *transaction fee*, то есть комиссия за транзакцию. Она является вторым по важности источником дохода для майнеров и именно от нее зависит время включения транзакции в блокчейн. Это связано с тем, что у каждого майнера существует некоторый пул непроверенных транзакций, которые претендуют на попадание в блок, и, как правило, майнер просто сортирует их по убыванию комиссии, тем самым максимизируя свою прибыль. Поэтому чем больше комиссия, тем выше вы окажетесь в очереди и тем быстрее пройдет ваш платеж.

На картинке ниже вы можете видеть фоторобота майнера, которому пришла [транзакция с комиссией в 135.000\\$](#).



UTXO

Как только новая транзакция занесена в блокчейн, ее выходы могут быть использованы в качестве входов. Для таких, пока еще непотраченных выходов, существует специальное название - **UTXO** (*unspent transaction output*). Как я уже говорил, каждый выход может быть использован в качестве входа только один раз, поэтому на практике интерес представляют именно непотраченные выходы, а уже использованные хранятся скорее как дань безопасности системы.

BTW под UTXO часто подразумевают весь массив непотраченных выходов, хотя воспитанные молодые люди должны писать *UTXO pool* ну или в крайнем случае *UTXO set*.

Возвращаясь к началу статьи, теперь вам должно быть понятно, что для подсчета баланса адреса не нужно перебирать весь блокчейн, а достаточно обойтись только перебором *UTXO pool*, что, очевидно, быстрее.

Structure

Общий вид транзакции описан в [официальной спецификации протокола](#), здесь же я приведу живой пример, взятый из блога [Ken Shirriff](#).

version	01 00 00 00
input count	01
input	previous output hash (reversed) 48 4d 40 d4 5b 9e a0 d6 52 fc a8 25 8a b7 ca a4 25 41 eb 52 97 58 57 f9 6f b5 0c d7 32 c8 b4 81
	previous output index 00 00 00 00
	script length
	scriptSig script containing signature
	sequence ff ff ff ff
output count	01
output	value 62 64 01 00 00 00 00 00
	script length
	scriptPubKey script containing destination address
block lock time	00 00 00 00

По какой-то [загадочной причине](#), *value* и *previous output hash* должны быть представлены в *little endian* форме, то есть в нашем случае хэш [транзакции](#), на которую мы ссылаемся, вообще-то равен *81 b4 c8 32...*, хотя в транзакции он записывается в виде ...*32 c8 b4 81*. Точно так же сумма транзакции равна *0.00091234* BTC или *0x016462* в hex, но в протоколе она записывается как *62 64 01 00 00 00 00 00*.

BTW хэш транзакции считается крайне просто - берете всю транзакцию в виде последовательности байт (в примере выше получается строка вида *010000000148....00*), два раза считаете от нее хэш SHA-256 и представляете результат в little endian форме.

previous output index - ссылаемся не на саму транзакцию, хэш которой указан в *previous output hash*, а на один из ее выходов. В этом параметре мы и указываем, какой конкретно выход нас интересует, нумерация начинается с нуля. Кстати, в тексте я часто буду говорить именно о "ссылке на транзакцию", но это только ради выразительности языка.

[*block lock time*](#) - этот параметр довольно редко используется на практике. Если он не равен 0 и меньше 500 млн, то это номер блока, начиная с которого данной транзакцией можно воспользоваться в качестве входа. Так как в среднем блоки появляются раз в 10 минут, то несложно прикинуть время, когда транзакция "откроется".

Если *lock time* больше 500 млн, то он означает UNIX timestamp, начиная с которого транзакция станет доступна. В нашем случае там стоит 0, то есть транзакция доступна сразу.

[*sequence*](#) - эта фича больше не используется, почитать про нее можно [здесь](#).

Параметры со словом *script* в названии существенно сложнее, о них будет рассказано ниже.

Script

Скорее всего вы уже слышали, что в сети Bitcoin существует механизм, основанный на криптостойких алгоритмах + паре приватный / публичный ключ, позволяющий создать систему, в которой только владелец приватного ключа может воспользоваться монетами, ассоциированными с адресом, полученным из этого ключа. Сейчас я покажу, как это реализуется "под капотом".

Начнем с того, что внутри Bitcoin существует свой собственный язык программирования, названный Script. Вот что о нем пишет [Bitcoin wiki](#):

Bitcoin uses a scripting system for transactions. Forth-like, Script is simple, stack-based, and processed from left to right. It is purposefully not Turing-complete, with no loops.

Суть в том, что язык прост как пробка, stack-based и Тьюринг-неполный. Вот пример типичной программы:

```
1 OP_DUP OP_DUP 5 OP_HASH160
```

Каждая инструкция называется *opcode* - [всего их порядка 80](#), так что язык действительно довольно примитивен. На картинке ниже изображен процесс исполнения программы `2 3 OP_ADD 5 OP_EQUAL`:

STACK	<p>SCRIPT</p> <hr/> <p>2 3 ADD 5 EQUAL</p> <hr/> <p style="text-align: center;">EXECUTION POINTER</p>
	<p>Execution starts from the left Constant value "2" is pushed to the top of the stack</p>
STACK	<p>SCRIPT</p> <hr/> <p>2 3 ADD 5 EQUAL</p> <hr/> <p style="text-align: center;">EXECUTION POINTER</p>
	<p>Execution continues, moving to the right with each step Constant value "3" is pushed to the top of the stack</p>
STACK	<p>SCRIPT</p> <hr/> <p>2 3 ADD 5 EQUAL</p> <hr/> <p style="text-align: center;">EXECUTION POINTER</p>
	<p>Operator ADD pops the top two items out of the stack and adds them together (3 add 2); then Operator ADD pushes the result (5) to the top of the stack</p>
STACK	<p>SCRIPT</p> <hr/> <p>2 3 ADD 5 EQUAL</p> <hr/> <p style="text-align: center;">EXECUTION POINTER</p>
	<p>Constant value "5" is pushed to the top of the stack</p>
STACK	<p>SCRIPT</p> <hr/> <p>2 3 ADD 5 EQUAL</p> <hr/> <p style="text-align: center;">EXECUTION POINTER</p>
	<p>Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5) and if they are equal, EQUAL pushes TRUE (TRUE = 1) to the top of the stack</p>

Lock & unlock transaction

Вернемся к языку чуть позже, а сначала давайте разберемся, зачем он здесь вообще нужен.

Для этого вспоминаем структуру транзакции и два параметра: *scriptSig* и *scriptPubKey*. В отличие от других параметров, назначение этих двух вообще не очевидно, и имхо это самое сложное, что есть в Bitcoin.

Я видел много попыток объяснить (как правило неудачных), что же из себя представляют скрипты в Bitcoin и как нужно их воспринимать на интуитивном уровне. Тем не менее я рискну и попробую привести еще одну аналогию. Для этого давайте рассмотрим *завещание*, вроде такого:

1.000.000\$ переходят к Алисе только после того, как ей исполнится 18 лет

В этом случае, сам текст завещания - это некоторое условие, при котором можно воспользоваться деньгами (читай **можно воспользоваться транзакцией на 1.000.000\$ как входом**), а ксерокопия паспорта в 19 лет - это доказательство того, что условие выполнено и самое время получить деньги.

Именно для того, чтобы **задать условие, при котором можно будет потратить выход, и для возможности подтвердить то, что условие выполнено** и нужен SCRIPT, приватные / публичные ключи и прочие сложности.

В случае Bitcoin, завещание - это *locking script*, который указывается в транзакции внутри поля [pk_script](#). Его еще часто называют *scriptPubKey* из-за того, что чаще всего это программа, содержащая публичный ключ или адрес, хотя, вообще говоря, он может не иметь ничего общего с криптографией.

Своего рода "доказательство" того, что условие из *locking script* выполнено, называется *unlocking script*, пишется в поле *signature script* и часто называется *scriptSig*, догадайтесь почему.

Сам механизм проверки скрипта на валидность очень прост - для этого нужно соединить *unlocking script* + *locking script* и запустить получившуюся программу как одно целое. Если после исполнения, сверху стека останется `TRUE`, то транзакция валидна, и невалидна в любом другом случае.

Multiplication-based script

Скорее всего, вы ничего не поняли, поэтому давайте напишем какой-нибудь максимально простой скрипт, чтобы окончательно во всем разобраться. Идея состоит в том, чтобы заблокировать деньги с помощью какого-нибудь числа, например `370`. *Locking script* будет выглядеть как `OP_MUL 370 OP_EQUAL` и для того, чтобы разблокировать транзакцию, нужно будет указать два числа, дающие 370 в произведении.

Для экспериментов со Script воспользуемся [онлайн площадкой](#) для запуска и дебага Bitcoin скриптов. В *unlocking script* запишем например `10 37`. [Проверяем](#):

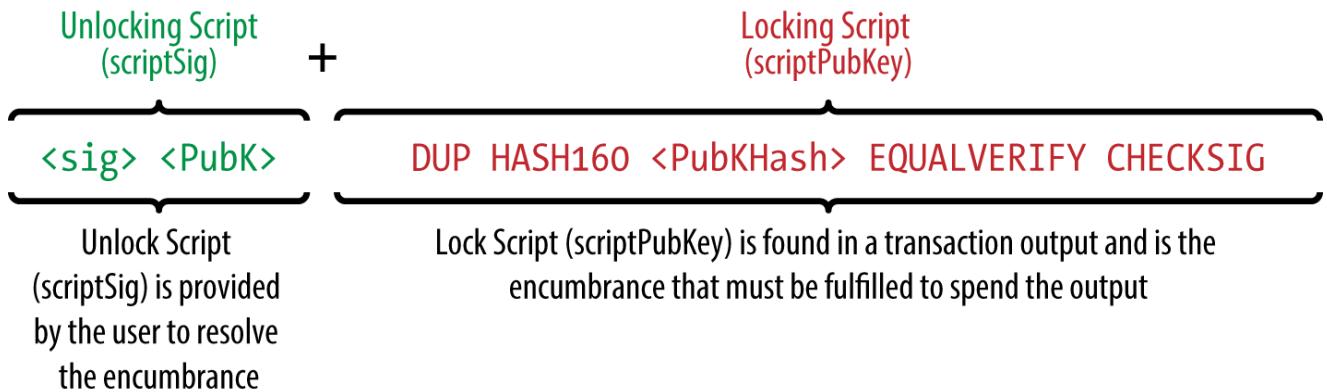
```
10 37 OP_MUL 370 OP_EQUAL
```

```
1 stack.push("10");
2 stack.push("37");
3 stack.OP_MUL();
4 stack.push("370");
5 stack.OP_EQUAL();
6 return stack.OP_VERIFY();
```



Pay to Public Key Hash (P2PKH)

P2PKH используется, наверное, в 99 транзакциях из 100, так что стоит понимать, как он работает. Вот его общий вид:

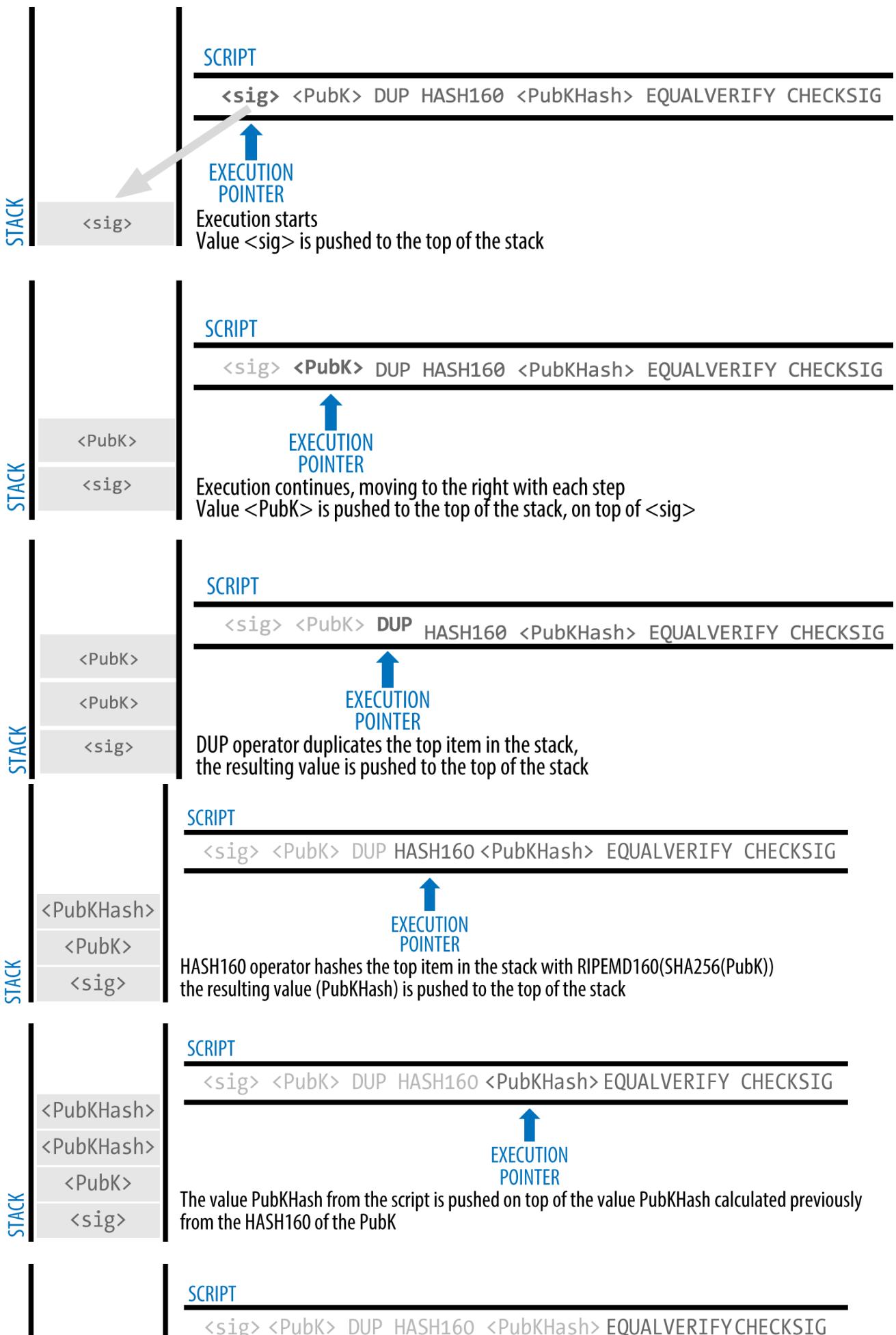


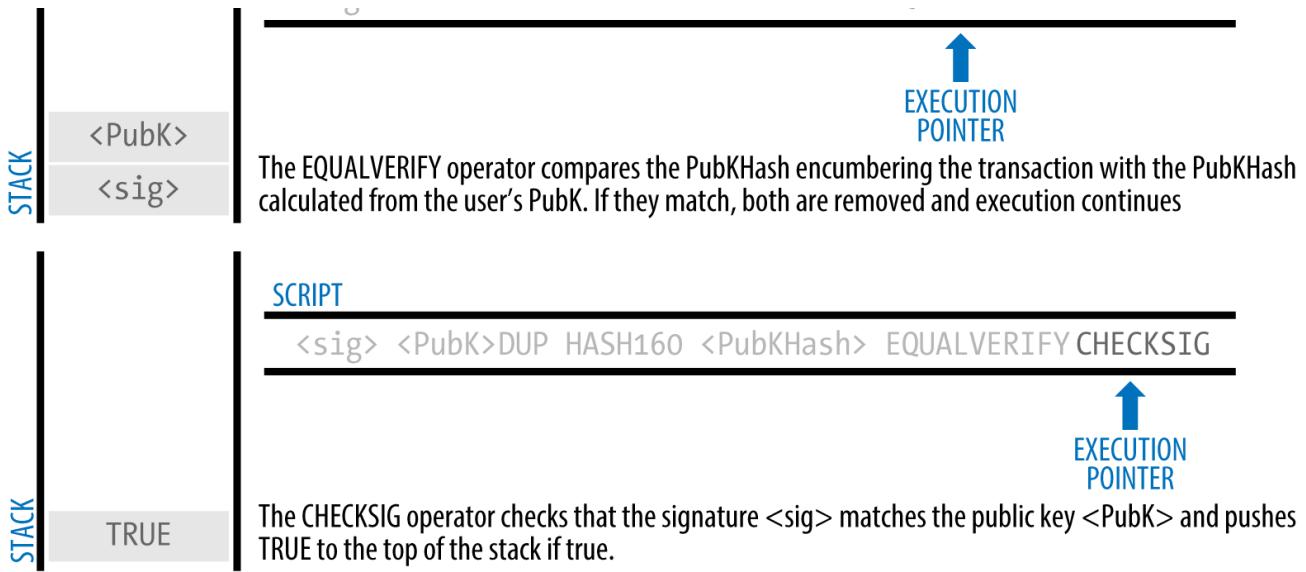
Этот скрипт известен с самого появления Bitcoin и, возможно, придуман самим Сатоши. Именно он выполняет ту задачу, о которой я писал выше: *сделать так, чтобы только владелец приватного ключа смог воспользоваться монетами, ассоциированными с адресом, полученным из этого ключа.*

На пальцах это выглядит следующим образом: пусть вашему другу **B** принадлежит приватный ключ **P**. Он получает из него публичный ключ **K**, адрес **A** и сообщает адрес вам. Далее вы отправляете на адрес **A** 1 BTC и в поле *locking script* пишите примерно следующее:

Только тот, кто владеет приватным ключом для адреса **A**, сможет потратить эту транзакцию. В качестве доказательства запишите в *unlocking script*, во-первых, публичный ключ **K**, а во-вторых подпись своей транзакции приватным ключом **P**.

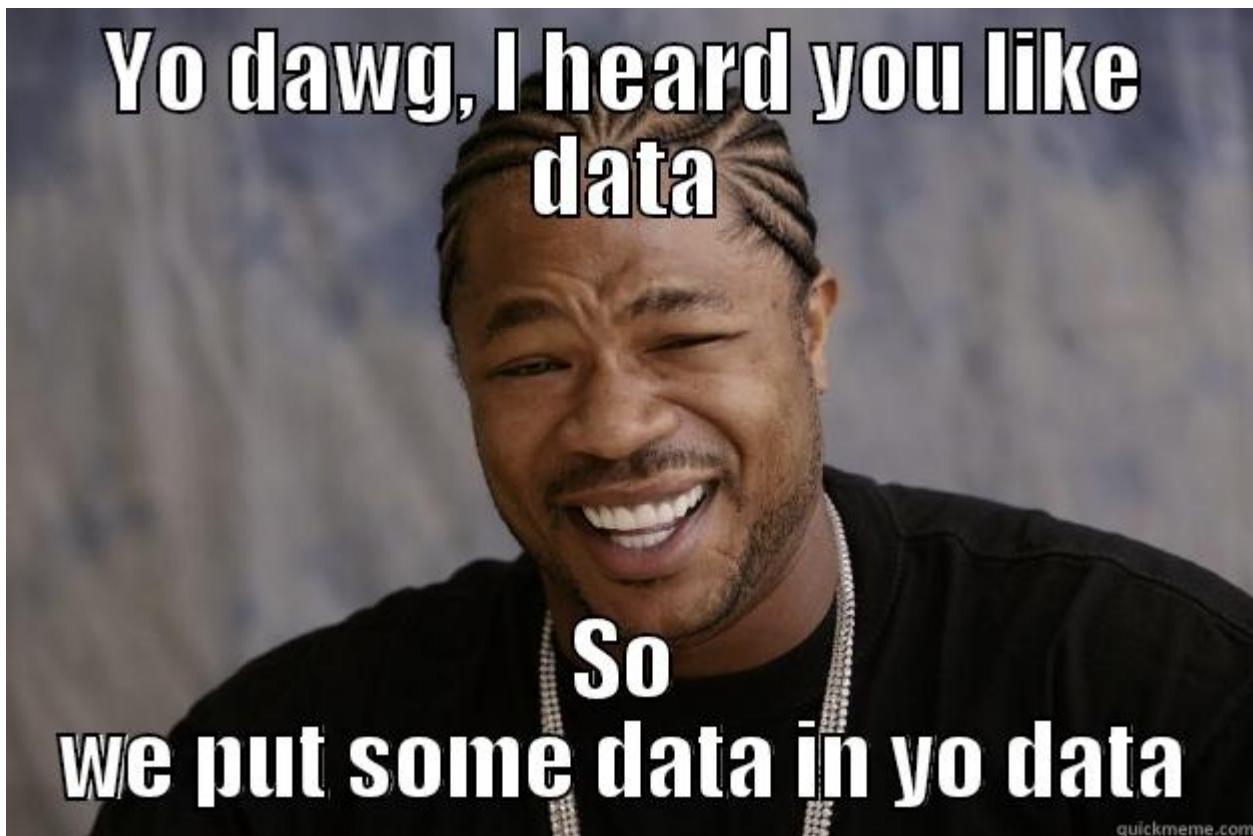
Когда **B** решит использовать вашу транзакцию в качестве входа, то он создаст свою, например, на 0.5 BTC, а в поле *unlocking script* вставит подпись своей транзакции приватным ключом **P** - `<sig>` и сам публичный ключ **K** - `<PubK>`.





1. Подпись транзакции добавляется в стек
2. Публичный ключ добавляется в стек
3. `OP_DUP` берет верхний элемент стека и дублирует его, теперь в стеке сверху два публичных ключа
4. `OP_HASH160` заменяет верхний элемент стека на его хэш `RIPEMD160(SHA256(x))`
5. В стек добавляется такой же хэш публичного ключа, но уже посчитанный отправителем транзакции. Если вы внимательно читали [Bitcoin in a nutshell - Cryptography](#), то для вас должно быть очевидно, что `RIPEMD160(SHA256(public_key))` и адрес - это в принципе одно и то же.
6. `OP_EQUALVERIFY` удаляет два верхних элемента стека, и если они не равны, то исполнение программы прерывается с ошибкой
7. `OP_CHECKSIG` проверяет подпись на соответствие транзакции. Если все верно, то удаляет подпись, удаляет публичный ключ и добавляет `TRUE`

P2P storage



Одно из самых интересных свойств Bitcoin, да и технологии блокчейн вообще, - это неизменяемость и гипотетическая "вечность" всего, что туда попадает. Неудивительно, что со временем нашлись люди, захотевшие использовать это в своих целях. И первое, что пришло им в голову - попытаться сохранить в блокчейн какие-нибудь сторонние данные и получить P2P дропбокс.

Я думаю вы уже поняли, как это делается. Берем строку `Make America great again` и просто записываем ее в *locking script*. Это все еще будет вполне корректный скрипт, другое дело, что к нему не получится придумать такой *unlocking script*, чтобы разблокировать средства. Но если вы отправите на выход с таким скриптом, условно говоря, 0.0000001 BTC, то в принципе и не жалко. Единственное ограничение - это размер вашей транзакции. Считайте, что она не может быть больше 100 КБ, хотя в реальности там все немного сложнее, можете почитать [здесь](#).

Понятное дело, что такое положение дел по душе не всем. У Bitcoin и так большие проблемы с масштабируемостью, а тут еще и блокчейн, без того немаленький, начинает засоряться всякими левыми данными. Более того, помним, что такие транзакции нельзя потратить, а значит они навсегда останутся в *UTXO pool*, что [ничуть не лучше](#).

Для того, чтобы достичь компромиса, был добавлен `OP_RETURN`, который позволяет "легально" хранить в блокчейне до 40 КБ данных.

OP_RETURN is a [script](#) opcode used to mark a transaction output as invalid. Since the data after `OP_RETURN` are irrelevant to Bitcoin payments, arbitrary data can be added into the output after an `OP_RETURN` - [Bitcoin wiki](#)

Вот так выглядит простейший *locking script* с его участием: `OP_RETURN <40kb data>`. Что примечательно, выход с таким скриптом приобретает статус *provably unspendable*, то есть *доказуемо непотрачиваемый*. Из-за этого он даже не попадает в *UTXO pool*, тем самым экономя драгоценное место. Остальные причины использовать `OP_RETURN <data>` вместо `<data>` вы можете найти [здесь](#).

Спойлер - их нет, если вы конечно не убежденный альтруист.

Links

- [The Bitcoin Script language \(pt. 1\)](#)
- [The Bitcoin Script language \(pt. 2\)](#)
- [Standart scripts](#)
- [Explanation of what an OP_RETURN transaction looks like](#)
- [The Bitcoin Script Playground](#)
- [NPM bitcoin-script package](#)

Транзакции - это чуть ли не самый "главный" объект в сети Bitcoin, да и в других блокчейнах тоже. Поэтому я решил, что если и писать про них целую главу, то тогда нужно рассказать и показать вообще все, что можно. В частности то, как они строятся и работают на уровне протокола.

Ниже я объясню, каким образом формируется транзакция, покажу как она подписывается и продемонстрирую механизм общения между нодами.

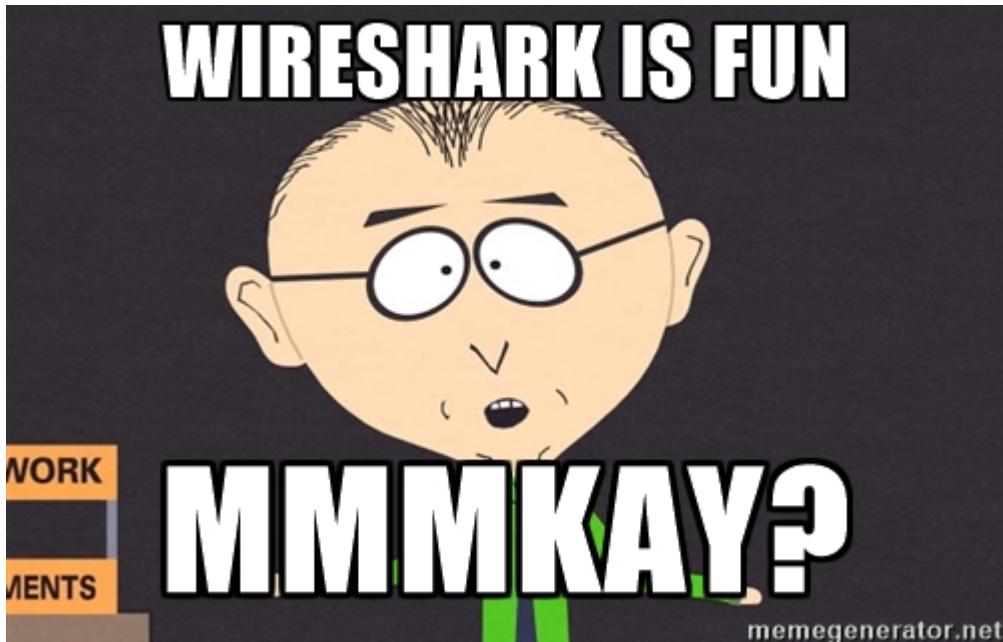


Table of content

1. Keys and address
2. Searching for nodes
3. Version handshake
4. Setting up a connection
5. Making transaction
6. Signing transaction
7. Sniff & spoof
8. Sending transaction
9. Links

Keys and address

Для начала создадим новую пару ключей и адрес. Как это делается я рассказывал в главе [Bitcoin in a nutshell - Cryptography](#), так что здесь все должно быть понятно. Для ускорения процесса возьмем вот этот [набор инструментов для Bitcoin](#), написанный самим [Виталиком Бутерином](#), хотя при желании вы можете воспользоваться уже написанными [фрагментами кода](#).

```

$ git clone https://github.com/vbuterin/pybitcointools
$ cd pybitcointools
$ sudo python setup.py install
$ python
Python 2.7.12 (default, Jul 1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from bitcoin import *
>>> private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"
>>> public_key = privtopub(private_key)
>>> public_key
'0497e922cac2c9065a0cac998c0735d9995ff42fb6641d29300e8c0071277eb5b4e770fcc086f322339bdefef4d5b51a
23d88755969d28e965daca5d0d2a0e09'
>>> address = pubtoaddr(public_key)
>>> address
'1LwPhYQi4BRBuuuWSGVeb6kPrTqpSVmoYz'

```

Я [скинул](#) на адрес `1LwPhYQi4BRBuuuWSGVeb6kPrTqpSVmoYz` 0.00012 BTC, так что теперь можно экспериментировать по полной программе.

Searching for nodes

Вообще говоря, это хорошая задача на подумать: как найти других участников сети при том, что сеть децентрализована? Подробнее про это можете почитать [здесь](#), скажу заранее, совсем децентрализованного решения пока что не существует.

Я покажу два способа. Первый - это *DNS seeding*. Суть в том, что есть некоторые доверенные адреса, такие как:

- bitseed.xf2.org
- dnsseed.bluematt.me
- seed.bitcoin.sipa.be
- dnsseed.bitcoin.dashjr.org
- seed.bitcoinstats.com

Они захардкожены в [chainparams.cpp](#) и командой `nslookup` можно получить от них адреса нод.

```

$ nslookup bitseed.xf2.org
Non-authoritative answer:
Name: bitseed.xf2.org
Address: 76.111.96.126
Name: bitseed.xf2.org
Address: 85.214.90.1
Name: bitseed.xf2.org
Address: 94.226.111.26
Name: bitseed.xf2.org
Address: 96.2.103.25
...

```

Другой способ не такой умный и на практике не используется, но в учебных целях он подходит даже лучше. Заходим на [Shodan](#), регистрируемся, авторизуемся и в строке поиска пишем `port:8333`. Это стандартный порт для `bitcoind`, в моем случае нашлось примерно 9.000 нод:

TOP COUNTRIES

Country	Count
United States	2,222
Germany	1,181
France	546
United Kingdom	540
Canada	467

70.68.73.137
S01066c3bb64e867.vf.shawcable.net
Shaw Communications
Added on 2016-12-26 16:38:58 GMT
CANADA, Coquitlam
[Details](#)

52.64.173.190
ec2-52-64-173-190.ap-southeast-2.compute.amazonaws.com
Amazon.com
Added on 2016-12-26 16:37:41 GMT
AUSTRALIA, Sydney
[Details](#)

93.233.206.19
p5DE9CE13.dip0.t-ipconnect.de
Deutsche Telekom AG
Added on 2016-12-26 16:33:50 GMT
GERMANY
[Details](#)

192.243.215.2
192-243-215-2.van0.pacificservers.com
Pacific Servers
Added on 2016-12-26 16:18:57 GMT
CANADA, Vancouver
[Details](#)

108.49.55.35
pool-108-49-55-35.bstnma.fios.verizon.net
Verizon FIOS
Added on 2016-12-26 16:13:02 GMT
UNITED STATES, Ashland
[Details](#)

185.81.165.156
15219.s.14vps.eu
UAB Interneto vizija
Added on 2016-12-26 16:10:32 GMT
ROMANIA
[Details](#)

Version handshake

Установка соединения между нодами начинается с обмена двумя сообщениями. Первым отправляется [version message](#), а в качестве ответа на него используется [verack message](#). Вот иллюстрация процесса *version handshake* из [Bitcoin wiki](#):

When the local peer **L** connects to a remote peer **R**, the remote peer will not send any data until it receives a version message.

- **L → R** Send version message with the local peer's version
- **R → L** Send version message back
- **R** Sets version to the minimum of the 2 versions
- **R → L** Send verack message
- **L** Sets version to the minimum of the 2 versions

Это делается в первую очередь для того, чтобы ноды узнали, какой версией протокола пользуется их "собеседник" и могли общаться на одном языке.

Setting up a connection



Каждое сообщение в сети [должно представляться](#) в виде `magic + command + lenght + checksum + payload`, за это отвечает функция `makeMessage`. Этой функцией мы еще воспользуемся, когда будем отправлять транзакцию.

В коде будет постоянно использоваться библиотека `struct`. Она отвечает за то, чтобы представлять параметры в правильном формате. Например `struct.pack("q", timestamp)` записывает текущее UNIX время в `long long int`, как этого и требует протокол.

```

import time
import socket
import struct
import random
import hashlib

def makeMessage(cmd, payload):
    magic = "F9BEB4D9".decode("hex") # Main network ID
    command = cmd + (12 - len(cmd)) * "\00"
    length = struct.pack("I", len(payload))
    check = hashlib.sha256(hashlib.sha256(payload).digest()).digest()[:4]
    return magic + command + length + check + payload

def versionMessage():
    version = struct.pack("i", 60002)
    services = struct.pack("Q", 0)
    timestamp = struct.pack("q", time.time())

    addr_recv = struct.pack("Q", 0)
    addr_recv += struct.pack(">16s", "127.0.0.1")
    addr_recv += struct.pack(">H", 8333)

    addr_from = struct.pack("Q", 0)
    addr_from += struct.pack(">16s", "127.0.0.1")
    addr_from += struct.pack(">H", 8333)

    nonce = struct.pack("Q", random.getrandbits(64))
    user_agent = struct.pack("B", 0) # Anything
    height = struct.pack("i", 0) # Block number, doesn't matter

    payload = version + services + timestamp + addr_recv + addr_from + nonce + user_agent +
height

    return payload

if __name__ == "__main__":
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("93.170.187.9", 8333))

    sock.send(makeMessage("version", versionMessage()))
    sock.recv(1024) # receive version message
    sock.recv(1024) # receive verack message

```

Теперь открываем Wireshark, ставим фильтр `bitcoin` или `tcp.port == 8333` и смотрим на получившиеся пакеты. Если все сделано верно, то, во-первых, будет верно определен протокол, *user-agent*, *block start height* и так далее. Во вторых, как и обещалось, вам прилетит ответ в виде сообщений *version* и *verack*. Теперь, когда соединение установлено, можно начинать работу.

75290 3597.475281	192.168.0.107	93.170.187.9	Bitcoin	163 version
75292 3597.547412	93.170.187.9	192.168.0.107	Bitcoin	184 version
75294 3597.634593	93.170.187.9	192.168.0.107	Bitcoin	430 verack, alert, ping
> Frame 75290: 163 bytes on wire (1304 bits), 163 bytes captured (1304 bits) on interface 0				
> Ethernet II, Src: Tp-LinkT_a5:8c:ac (64:70:02:a5:8c:ac), Dst: Tp-LinkT_a5:8c:ac (64:70:02:a5:8c:ac)				
> Internet Protocol Version 4, Src: 192.168.0.107, Dst: 93.170.187.9				
> Transmission Control Protocol, Src Port: 50137, Dst Port: 8333, Seq: 1, Ack: 1, Len: 109				
▼ Bitcoin protocol				
Packet magic: 0xf9beb4d9				
Command name: version				
Payload Length: 85				
Payload checksum: 0xcc444863				
▼ Version message				
Protocol version: 60002				
> Node services: 0x0000000000000000				
Node timestamp: Dec 30, 2016 17:10:50.000000000 Russia TZ 2 Standard Time				
> Address as receiving node				
> Address of emmitting node				
Random nonce: 0x9408a4e1df9c2216				
▼ User agent				
Count: 0				
String value:				
Block start height: 0				

Making transaction

Перед созданием транзакции еще раз открываем [спецификацию](#) и внимательно ее придерживаемся. Отклонение на 1 байт уже делает транзакцию невалидной, так что нужно быть предельно аккуратным.

Для начала зададим адреса, приватный ключ и хэш [транзакции](#), на которую мы будем ссылаться:

```
previous_output = "60ee91bc1563e44866c66937b141e9ef4615a272fa9d764b9468c2a673c55e01"
receiver_address = "1C29gpF5MkEPrECiGtKVXwWdAmNiQ4PBMH"
my_address = "1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz"
private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"
```

Далее создадим транзакцию в *raw* виде, то есть пока что неподписанную. Для этого достаточно просто следовать спецификации:

```

def txnMessage(previous_output, receiver_address, my_address, private_key):
    receiver_hashed_pubkey= base58.b58decode_check(receiver_address)[1:].encode("hex")
    my_hashed_pubkey = base58.b58decode_check(my_address)[1:].encode("hex")

    # Transaction stuff
    version = struct.pack("<L", 1)
    lock_time = struct.pack("<L", 0)
    hash_code = struct.pack("<L", 1)

    # Transactions input
    tx_in_count = struct.pack("<B", 1)
    tx_in = {}
    tx_in["outpoint_hash"] = previous_output.decode('hex')[::-1]
    tx_in["outpoint_index"] = struct.pack("<L", 0)
    tx_in["script"] = ("76a914%s88ac" % my_hashed_pubkey).decode("hex")
    tx_in["script_bytes"] = struct.pack("<B", (len(tx_in["script"])))
    tx_in["sequence"] = "ffffffff".decode("hex")

    # Transaction output
    tx_out_count = struct.pack("<B", 1)

    tx_out = {}
    tx_out["value"]= struct.pack("<Q", 1000) # Send 1000 satoshis
    tx_out["pk_script"]=( "76a914%s88ac" % receiver_hashed_pubkey).decode("hex")
    tx_out["pk_script_bytes"] = struct.pack("<B", (len(tx_out["pk_script"])))

    tx_to_sign = (version + tx_in_count + tx_in["outpoint_hash"] + tx_in["outpoint_index"] +
                  tx_in["script_bytes"] + tx_in["script"] + tx_in["sequence"] + tx_out_count +
                  tx_out["value"] + tx_out["pk_script_bytes"] + tx_out["pk_script"] + lock_time +
                  hash_code)

```

Заметьте, что в поле `tx_in["script"]` написано отнюдь не `<Sig> <PubKey>`, как вы, наверное, ожидали. Вместо этого указан **блокирующий скрипт выхода, на который мы ссылаемся**, в нашем случае это `OP_DUP OP_HASH160 dab3cccc50d7ff2d1d2926ec85ca186e61aef105 OP_EQUALVERIFY OP_CHECKSIG`.

BTW нет никакой разницы между привычным `OP_DUP OP_HASH160`
`dab3cccc50d7ff2d1d2926ec85ca186e61aef105 OP_EQUALVERIFY OP_CHECKSIG` и
`76a914dab3cccc50d7ff2d1d2926ec85ca186e61aef105s88ac` - во втором случае просто используется
[специальная кодировка для экономии места](#):

```

0x76 = OP_DUP
0xa9 = OP_HASH160
0x14 = далее следует 14 байт информации
dab3cccc50d7ff2d1d2926ec85ca186e61aef105s88ac
...

```

Signing transaction

Теперь самое время подписать транзакцию, здесь все довольно просто:

```

hashed_raw_tx = hashlib.sha256(hashlib.sha256(tx_to_sign).digest()).digest()
sk = ecdsa.SigningKey.from_string(private_key.decode("hex"), curve=ecdsa.SECP256k1)
vk = sk.verifying_key
public_key = ('\x04' + vk.to_string()).encode("hex")
sign = sk.sign_digest(hashed_raw_tx, sigencode=ecdsa.util.sigencode_der)

```

После того, как получена подпись для *raw transaction*, можно заменить *unlocking script* на настоящий и привести транзакцию к окончательному виду:

```

sigscript = sign + "\x01" + struct.pack("<B", len(public_key.decode("hex"))) +
public_key.decode("hex")

real_tx = (version + tx_in_count + tx_in["outpoint_hash"] + tx_in["outpoint_index"] +
struct.pack("<B", (len(sigscript) + 1)) + struct.pack("<B", len(sign) + 1) + sigscript +
tx_in["sequence"] + tx_out_count + tx_out["value"] + tx_out["pk_script_bytes"] +
tx_out["pk_script"] + lock_time)

return real_tx

```

Sniff & spoof

Здесь нужно пояснить одну деталь. Я думаю вы понимаете, зачем мы вообще подписываем транзакции. Это делается для того, чтобы никто не смог изменить наше сообщение и отправить его дальше по сети, потому что изменится подпись сообщения и так далее.

Но если вы внимательно читали, то запомнили, что мы подписываем ненастоящую транзакцию, которая в конечном итоге будет отправлена другим нодам, а ее модификацию, где в *unlocking script* указан *locking script* из выхода, на который мы ссылаемся. В принципе понятно, почему это происходит: в настоящий *unlocking script* должна быть записана эта самая подпись, и получается замкнутый круг: для правильной подписи нужен правильный *unlocking script*, для правильного *unlocking script* нужна правильная подпись. Так что Сатоши пошел на компромисс и разрешил пользоваться не совсем "настоящими" подписями.

Поэтому может случиться так, что кто-нибудь в сети поймает наше сообщение, изменит *unlocking script* и отправит отредактированное сообщение дальше. Никто из нод не сможет этого проверить, потому что подпись не "защищает" *unlocking script*. Эта уязвимость называется **Transaction malleability**, подробнее про нее вы можете почитать [здесь](#) или посмотреть доклад с Black Hat USA 2014 - [Bitcoin Transaction Malleability Theory in Practice](#).

TL;DR Если вы пользуетесь стандартными скриптами вроде P2PKH, то вам ничего не грозит. В противном случае стоит быть аккуратным.

Sending transaction

Отправка транзакции в сеть производится точно так же, как и в случае с *version message*:

```
if __name__ == "__main__":
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("70.68.73.137", 8333))

    sock.send(makeMessage("version", versionMessage()))
    sock.recv(1024) # version
    sock.recv(1024) # verack

    # Transaction options
    previous_output = "60ee91bc1563e44866c66937b141e9ef4615a272fa9d764b9468c2a673c55e01"
    receiver_address = "1C29gpF5MkEPrECiGtkVXwWdAmNiQ4PBMH"
    my_address = "1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz"
    private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"

    txn = txnMessage(previous_output, receiver_address, my_address, private_key)
    print "Signed txn:", txn

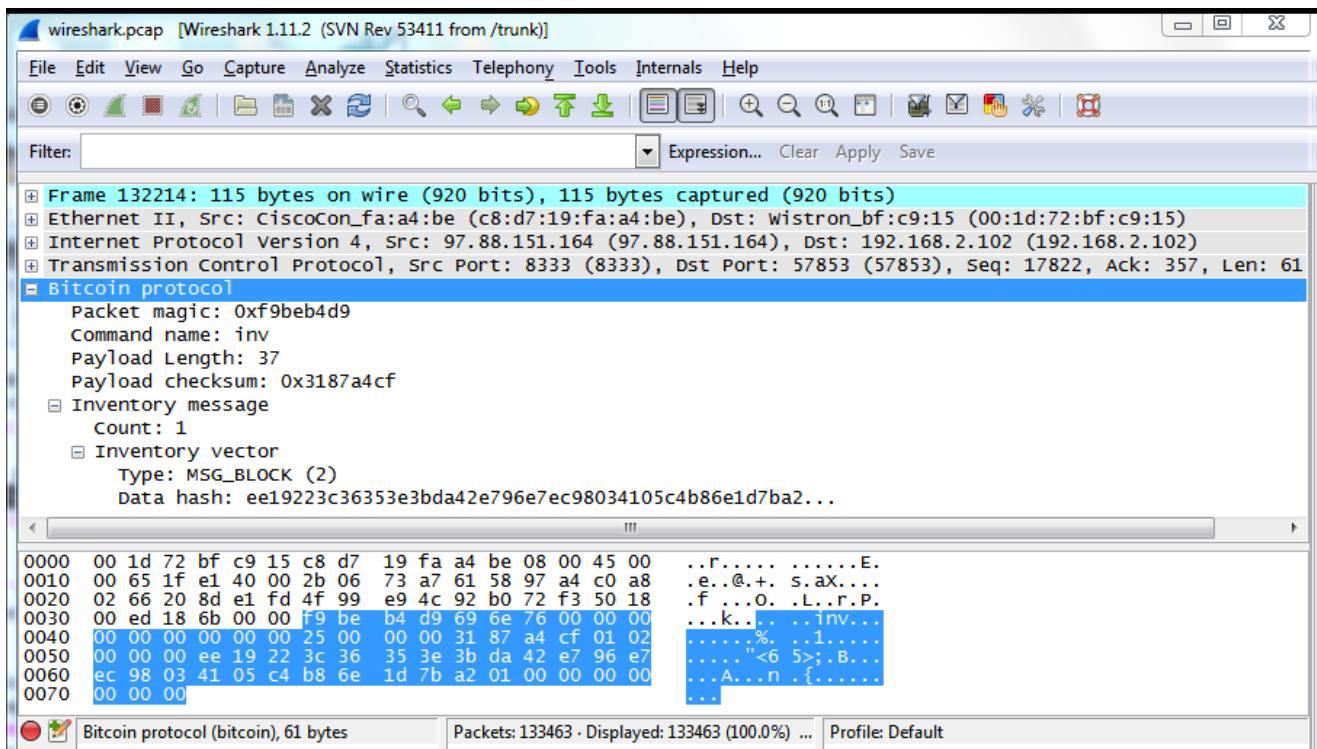
    sock.send(makeMessage("tx", txn))
    sock.recv(1024)
```

Запускаем получившийся код и бежим смотреть на пакеты. Если все сделано верно, то в качестве ответа на ваше сообщение придет [inv message](#) (в противном случае был бы [reject message](#)). Интересный факт - каждая нода, при получении свежей транзакции проверяет ее на валидность (процесс описан в [Bitcoin in a nutshell - Mining](#)), поэтому если вы где-то ошиблись, то вас об этом мгновенно оповестят:

26	2.101161	192.168.0.107	70.68.73.137	Bitco...	302	tx											
28	3.198843	70.68.73.137	192.168.0.107	Bitco...	115	inv											
▼ Bitcoin protocol																	
Packet magic: 0xf9beb4d9																	
Command name: tx																	
Payload Length: 224																	
Payload checksum: 0x69d32a8c																	
▼ Tx message																	
Transaction version: 1																	
Input Count: 1																	
▼ Transaction input																	
> Previous output																	
Script Length: 139																	
Signature script: 48304502210086fb3df5d4cc282649817051190536eaa2bb...																	
Sequence: 4294967295																	
Output Count: 1																	
▼ Transaction output																	
Value: 1000																	
Script Length: 25																	
Script: 76a91478e10cf8e4bd38266d8fd4ed5c8b430d30a3cde888...																	
Block lock time or block ID: 0																	
0000	64	70	02	a5	8c	ac	00	1d	60	1b	8f	be	08	00	45	00	dp.....`.....E.
0010	01	20	67	bc	40	00	80	06	00	00	c0	a8	00	6b	46	44	.g.@....kFD
0020	49	89	e7	b6	20	8d	40	b2	f9	f3	86	14	f3	a4	50	18	I...@.....P.
0030	00	ff	51	f3	00	00	f9	be	b4	d9	74	78	00	00	00	00	..Q.....tx....
0040	00	00	00	00	00	00	e0	00	00	00	69	d3	2a	8c	01	00i.*...
0050	00	00	01	01	5e	c5	73	a6	c2	68	94	4b	76	9d	fa	72^s..h.Kv..r
0060	a2	15	46	ef	e9	41	b1	37	69	c6	66	48	e4	63	15	bc	..F..A.7 i.fH.c..
0070	91	ee	60	00	00	00	8b	48	30	45	02	21	00	86	fb	..`.....H0E.!...	
0080	3d	f5	d4	cc	28	26	49	81	70	51	19	05	36	ea	a2	bb	=...(&I. pQ..6...
0090	20	60	4d	77	9a	88	54	96	4d	35	2b	f7	79	77	02	20	`Mw..T. M5+.yw.
00a0	38	7f	5d	e9	0f	42	f3	45	86	17	88	d3	47	bd	f6	bb	8.]..B.E....G...
00b0	82	7c	02	88	88	f3	72	95	f1	56	f7	bc	a1	b2	d6	f7r..V.....
00c0	01	41	04	97	e9	22	ca	c2	c9	06	5a	0c	ac	99	8c	07	.A...."....Z.....
00d0	35	d9	99	5f	f4	2f	b6	64	1d	29	30	0e	8c	00	71	27	5..._./d.)0...q'
00e0	7e	b5	b4	e7	70	fc	c0	86	f3	22	33	9b	de	fe	f4	d5	~...p...."3.....
00f0	b5	1a	23	d8	87	55	96	9d	28	e9	65	da	ca	aa	5d	0d	...#.U..(.e....].

Уже через несколько секунд после отправления транзакции в сеть, ее можно будет [отследить](#), правда сначала она будет числиться неподтвержденной. Потом, спустя какое-то время (вплоть до нескольких часов), транзакция будет включена в блок.

Если вы к тому времени не закроете Wireshark плюс в сообщении *version* укажете текущую высоту блокчейна, то вам прийдет уведомление о новом блоке в виде все того же *inv message*, но на этот раз с `TYPE = MSG_BLOCK` (я его закрыл, поэтому ниже скриншот из блога [Ken Shirriff](#)):



В **Data hash** вы можете видеть длинную строку, которая на самом деле является заголовком нового блока в *little endian* форме. В данном случае это блок [#279068](#) с заголовком `0000000000000001a27b1d6eb8c405410398ece796e742da3b3e35363c2219ee`. Куча ведущих нулей - не случайность, а результат майнинга, о котором я расскажу отдельно.

Но перед этим вам нужно разобраться с самим блокчейном, блоками, их заголовками и так далее. Поэтому следующая глава: [Bitcoin in a nutshell - Blockchain](#)

Links

- [Bitcoins the hard way: Using the raw Bitcoin protocol](#)
- [Analyzing Bitcoin Network Traffic Using Wireshark](#)
- [How the Bitcoin protocol actually works](#)
- [How does a Bitcoin node find its peers?](#)
- [Bitcoin Developer Reference. P2P network](#)
- [Redeeming a raw transaction step by step](#)

Blockchain - это технология, на базе которой построен Bitcoin. И если пару лет назад вся слава доставлась криптовалюте, то сегодня все чаще можно слышать смелые [фразы](#) вроде: "Forget Bitcoin, Long Live Blockchain". Активно развиваются платформы вроде Ethereum, IPFS или Overstock, которые рассматривают блокчейн не как инструмент для создания еще одной платежной системы, а как совершенно обособленную технологию, сравнимую по своей инновационности разве что с Интернетом.

В этой главе я расскажу вам, что из себя представляет блокчейн Bitcoin. Даже по сравнению с Ethereum, это жуткий анахронизм, но понимание принципов его работы вам сильно поможет, если вы решите разобраться с более сложными проектами.

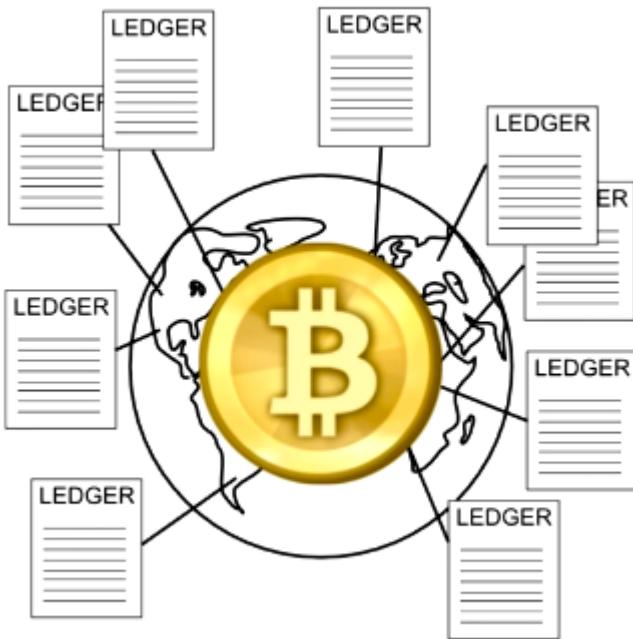


Table of content

1. Blockchain for dummies
2. Structure
3. Merkle tree
4. Timestamp
5. Raw block
6. Links

Blockchain for dummies

Сам по себе блокчейн - это крайне простая штука. Его проще всего проиллюстрировать на примере книги бухгалтерского учета, в которую записываются все транзакции в сети Bitcoin. Более того, такая книга присутствует у каждого участника сети, а значит любой, при желании, может проверить, была ли или иная транзакция в реальности или нет.



И если блокчейн целиком - это книга, то отдельные блоки можно представлять как страницы, на которых "записываются" транзакции. Каждый блок "ссылается" на предыдущий и так до самого первого блока (*genesis block*). Именно это и создает такую интересную особенность блокчейна, как неизменяемость. Нельзя взять и изменить блок #123 так, чтобы этого никто не заметил. Потому что блокчейн устроен таким образом, что это повлечет изменение блока #124, потом #125 и так далее, до самого верха.

Structure

Привычным движением руки открываем [спецификацию протокола](#) и смотрим на структуру блока.

Field Size	Description	Data type	Comments
4	version	int32_t	Block version information (note, this is signed)
32	prev_block	char[32]	The hash value of the previous block this particular block references
32	merkle_root	char[32]	The reference to a Merkle tree collection which is a hash of all transactions related to this block
4	timestamp	uint32_t	A Unix timestamp recording when this block was created (Currently limited to dates before the year 2106!)
4	bits	uint32_t	The calculated difficulty target being used for this block
4	nonce	uint32_t	The nonce used to generate this block... to allow variations of the header and compute different hashes
?	txn_count	var_int	Number of transaction entries
?	txns	tx[]	Block transactions, in format of "tx" command

- *version* - [версия](#) блока
- *prev_block* - хэш предыдущего блока (*parent block*)
- *merkle_root* - если упрощенно, то это хэш всех транзакций в блоке
- *timestamp* - дата и время создания блока
- *bits, nonce* - про эти параметры я подробно расскажу в главе [Bitcoin in a nutshell - Mining](#)
- *txn_count, txns* - число транзакций в блоке и их список

Первые шесть параметров (все кроме *txn_count* и *txns*) образуют заголовок блока (*header*). Именно хэш заголовка называют хэшем блока, то есть сами транзакции непосредственного участия в хэшировании не принимают.

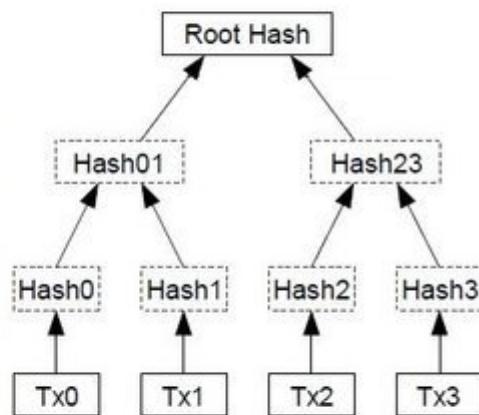
Вместо этого они заносятся в особую структуру - дерево Меркля, про которую я расскажу ниже.

Merkle tree

Technical side

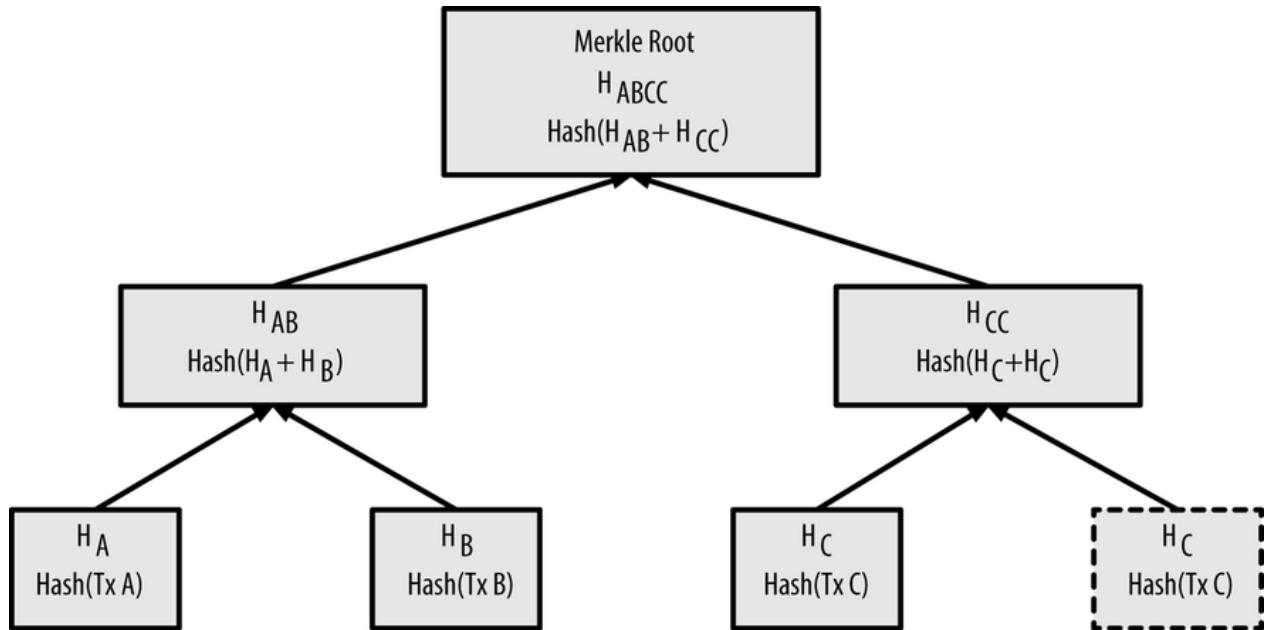


Merkle Tree



Дерево Меркля - это структура данных, также известная как бинарное дерево хэшей. В случае Bitcoin оно строится следующим образом:

1. Сначала считаются хэши всех транзакций в блоке $\text{hash_A} = \text{SHA256}(\text{SHA256}(A))$
2. Потом считаются хэши от суммы хэшей транзакций $\text{hash_AB} = \text{SHA256}(\text{SHA256}(\text{hash_A} + \text{hash_B}))$
3. Точно также считаем хэши от суммы получившихся хэшей $\text{hash_ABCD} = \text{SHA256}(\text{SHA256}(\text{hash_AB} + \text{hash_CD}))$ и далее по рекурсии. Лирическое отступление - так как дерево бинарное, то на каждом шаге должно быть четное число элементов. Поэтому если, например, у нас только три транзакции, то последняя транзакция просто дублируется:



4. Процесс продолжается до тех пор, пока не получится один единственный хэш - он и называется *merkle_root* (третье поле в *header* блока)

Ниже приведена реализация дерева Меркля, можете проверить ее на каком-нибудь блоке.

```

import hashlib

# Hash pairs of items recursively until a single value is obtained
def merkle(hashList):
    if len(hashList) == 1:
        return hashList[0]
    newHashList = []
    # Process pairs. For odd length, the last is skipped
    for i in range(0, len(hashList)-1, 2):
        newHashList.append(hash2(hashList[i], hashList[i+1]))
    if len(hashList) % 2 == 1: # odd, hash last item twice
        newHashList.append(hash2(hashList[-1], hashList[-1]))
    return merkle(newHashList)

def hash2(a, b):
    # Reverse inputs before and after hashing
    # due to big-endian / little-endian nonsense
    a1 = a.decode('hex')[::-1]
    b1 = b.decode('hex')[::-1]
    h = hashlib.sha256(hashlib.sha256(a1 + b1).digest()).digest()
    return h[::-1].encode('hex')

```

Immutability



Теперь о том, зачем это нужно в Bitcoin. Я думаю, вы понимаете, что если изменить хотя бы одну транзакцию, то *merkle_root* также изменится. Поэтому такая структура данных позволяет обеспечить "неподделываемость" транзакций в блоке. То есть не может произойти следующей ситуации:

Кто-то из майнеров нашел новый блок и начал раскидывать его по сети. В это время злоумышленник перехватывает блок и, например, удаляет из блока какую-нибудь транзакцию, после чего распространяет уже измененный блок.

Для проверки достаточно посчитать *merkle_root* самостоятельно и сравнить его с тем, что записан в *header* блока.

SPV

Но здесь можно резонно возразить, что, во-первых, такие сложности совершенно ни к чему.

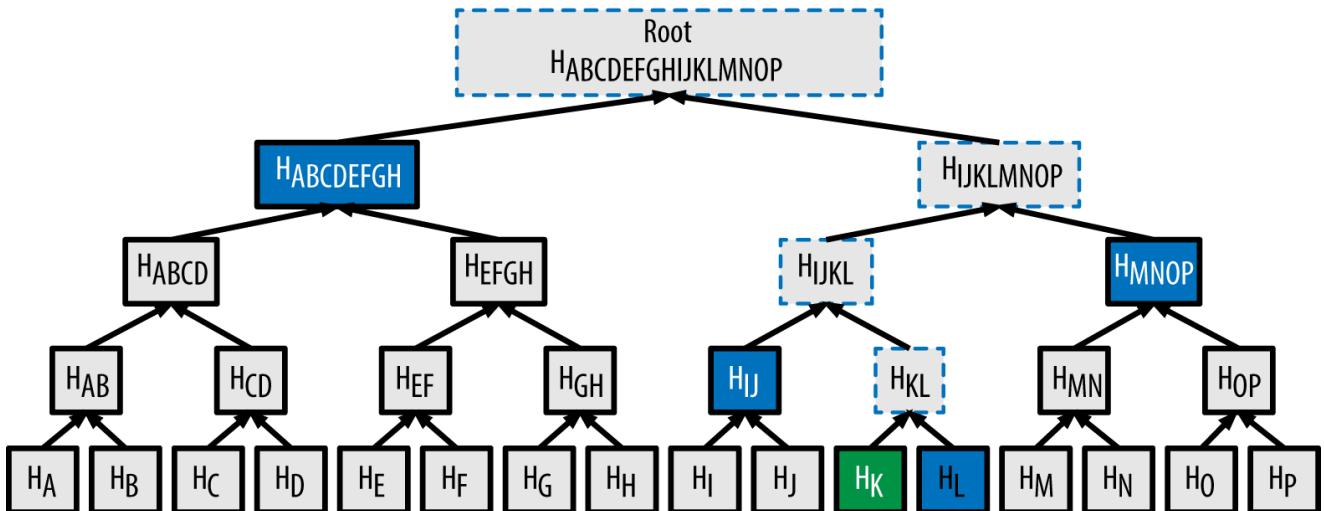
Достаточно просто посчитать хэш от суммы всех транзакций в блоке `txns_hash = SHA256(SHA256(sum(txns)))` - он точно также изменится после любых манипуляций с транзакциями. А, во-вторых, что мешает злоумышленнику подменить *merkle_root* в блоке? На второй вопрос отвечу сразу: на самом деле в блоке вообще нельзя ничего изменить, потому что блок тут же станет невалидным (это вы поймете после прочтения следующей главы [Bitcoin in a nutshell - Mining](#)).

А дерево Меркля нужно на самом деле для того, чтобы иметь возможность создавать *SPV nodes* (*Simplified Payment Verification*). Такие ноды синхронизируют только заголовки блоков, без самих транзакций. В результате блокчейн занимает на порядок меньше места (для красоты возьмем высоту в 500.000 блоков, размер *header* фиксирован - 80 байт):

$500.000 * 80 / 1024 / 1024 \approx 40 \text{ МБ}$

Такой блокчейн уже можно без проблем уместить на телефоне, планшете или каком-нибудь IoT. Что в перспективе должно дать большую децентрализацию, безопасность сети и так далее.

Суть упрощенной верификации платежей в следующем: пусть у вас есть *SPV* нода. У меня же есть весь блокчейн целиком и мне нужно вас убедить, что какая-нибудь транзакция действительно была (на картинке это транзакция *K*). В этом случае, мне достаточно всего лишь предоставить вам несколько хэшей: `H_L, H_IJ, H_MNOP, H_ABCDEFGH`, они еще называются *authentication path*.



После чего вы сначала считаете $H_K = \text{SHA256}(\text{SHA256}(K))$, потом $H_{KL} = \text{SHA256}(H_K + H_L)$ и так до самого верха. Если в итоге вы находитите у себя блок с таким же *merkle_root*, то факт существования транзакции считается подтвержденным.

BTW Ральф Меркл даже запатентовал свою структуру данных, о чем свидетельствует патент [US4309569 A](#).

Timestamp

Еще один интересный вопрос. Представим, что где-то в сети появился новый блок и ноды начинают передавать его друг-другу. Каждая нода должна убедиться в том, что блок корректен. Для этого она:

- проверяет синтаксис и структуру блока
- проверяет на валидность каждую транзакцию в блоке
- хэширует транзакции и сравнивает *merkle root*
- проверяет несколько критериев, связанных с майнингом, и так далее

Но как можно проверить timestamp? Понятно, что время на разных компьютерах может различаться, так что даже если у нового блока timestamp отличается от вашего текущего времени на час вперед, это еще не значит, что блок "неправильный", может у майнера просто часы спешат.

Поэтому для проверки timestamp на валидность было придумано [два критерия](#). Во-первых, он должен быть больше, чем среднее арифметическое timestamp-ов предыдущих 11 блоков. Это делается для того, чтобы не получилось так, что блок #123 вышел 12 марта 2011 года, а #124 - 13 февраля 1984. Но в тоже время допускается некоторая погрешность.

Во-вторых, timestamp должен быть меньше чем *network adjusted time*. То есть нода, при получении нового блока, интересуется текущим временем у своих "соседей" по сети, считает среднее арифметическое и если block timestamp меньше получившегося значения + 2 часа, то все в порядке.

BTW как вы видите, timestamp нового блока может оказаться даже меньше, чем timestamp более раннего блока. Это не такая уж и редкость, например [#145045](#), [#145046](#) и [#145047](#).

```
145044: 2011-09-12 15:46:39
145045: 2011-09-12 16:05:07
145046: 2011-09-12 16:00:05 // ~5 minutes before prior block
145047: 2011-09-12 15:53:36 // ~7 & ~12 minutes before 2 prior blocks
145048: 2011-09-12 16:04:06 // after 2 prior blocks but still before 145045
```

Raw block

Если у вас до сих остались какие-то вопросы по структуре блока, то предлагаю вам посмотреть на них в "сыром" виде. Самый очевидный способ это сделать - запустить на пару часов `bitcoind --daemon`, а потом исследовать уже скачанные блоки. Но, во-первых, не у всех есть время / желание синхронизировать блокчейн. Во-вторых, в Bitcoin блоки хранятся в крайне специфической базе данных [LevelDB](#), еще и довольно [странным образом](#). А так как книга расчитана не только на опытных разработчиков, то я пойду уже проверенным путем и снова использую протокол в его первозданном виде.

Для получения блока отправим сообщение [getdata](#), в котором укажем `type : MSG_BLOCK` и `hash : 000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1af33e506` - это хэш блока [#100.000](#). Весь код целиком можете посмотреть [здесь](#).

```
def getdataMessage():
    block_hash = '000000000003ba27aa200b1cecaad478d2b00432346c3f1f3986da1af33e506'

    count = struct.pack("<B", 1)
    inventory = struct.pack("<L", 2) # type : MSG_BLOCK
    inventory += block_hash.decode('hex')[::-1]

    return count + inventory
```

✓	16	8.116111	192.168.0.107	70.68.73.137	Bit...	115	getdata
	17	8.313963	70.68.73.137	192.168.0.107	Bit...	10...	block

```
· Bitcoin protocol
  Packet magic: 0xf9beb4d9
  Command name: block
  Payload Length: 957
  Payload checksum: 0x56324b0b
  ▾ Block message
    Block version: 1
    Previous block: 50120119172a610421a6c3011dd330d9df07b63616c2cc1f...
    Merkle root: 6657a9252aacd5c0b2940996ecff952228c3067cc38d4885...
    Block timestamp: Dec 29, 2010 14:57:43.000000000 Russia TZ 2 Standard Time
    Bits: 0x1b04864c
    Nonce: 0x10572b0f
    Number of transactions: 4
  ▾ Tx message [ 1 ]
    Transaction version: 1
    Input Count: 1
    ▶ Transaction input
    Output Count: 1
    ▶ Transaction output
    Block lock time or block ID: 0
```

Links

- [Mastering Bitcoin - The Blockchain](#)
- [Documentation of the physical Bitcoin blockchain](#)
- [What are the keys used in the blockchain levelDB](#)
- [bitcoin-core/leveldb/doc/table_format.txt](#)
- [Деревья Меркля в Эфириуме](#)

Даже люди, бесконечно далекие от темы криптовалют, скорее всего слышали про майнинг. Наверное и ты, дорогой читатель, задумывался о том, чтобы включить свой игровой Pentium 4 на ночь, а утром проснуться уже богатым.

Но, как это часто случается в мире блокчейна, тех кто слышал - много, а вот тех, кто реально понимает процесс от начала до конца, - единицы. Поэтому в последней главе я постарался максимально подробно охватить все тонкости, начиная от технической реализации PoW, заканчивая рентабельностью майнинга на видеокартах.



Table of content

1. Explain me like I'm five
2. Sky is the limit?
3. Reward
4. Hard challenge
5. Technical side
6. 2 Blocks 1 Chain
7. Hardware
8. Conclusion
9. Links

Explain me like I'm five

Майнинг, также **добыча** (от [англ.](#) *mining* — добыча полезных ископаемых) — деятельность по поддержанию распределенной платформы и созданию новых блоков с возможностью получить вознаграждение в форме эмитированной валюты и комиссионных сборов в различных [криптовалютах](#), в частности в [Биткойн](#). Производимые вычисления требуются для обеспечения защиты от повторного расходования одних и тех же единиц валюты, а связь майнинга с эмиссией стимулирует людей расходовать свои вычислительные мощности и поддерживать работу сетей - [Wikipedia](#)

Если на пальцах, то майнинг - это критически важный для Bitcoin процесс, состоящий в создании новых блоков и преследующий сразу две цели. Первая - производство денежной массы. Каждый раз, когда майнер создает новый блок, ему за это полагается награда в N-ое число монет, которые он потом где-нибудь тратит, тем самым запуская в сеть новые средства.

Вторая, и куда более важная цель, - обеспечение работы всей сети. Наверняка, читая предыдущие статьи, вы уже задавали себе вопросы "**Кто тот человек, который проверяет скрипты транзакций?**" или "**Если в качестве входа я укажу уже использованный выход, в какой момент это заметят?**".

Так вот, все эти действия выполняют в первую очередь майнеры. Ну, на самом деле каждый участник сети в той или иной степени обеспечивает ее безопасность. Синхронизировать Bitcoin так долго не потому что приходится качать 100 ГБ, а потому что надо проверить каждый байт, посчитать каждый хэш, запустить каждый скрипт и так далее.

Но если нарисовать весь процесс, начиная с нажатия кнопки "Send" в кошельке и заканчивая просмотром блока с вашей транзакцией где-нибудь на [blockchain.info](#), то именно майнеры будут решать, окажется ваша транзакция в блоке или нет.

Sky is the limit?



Для начала давайте еще раз пройдемся по первому пункту и обсудим понятие денежной массы.

Одна из фундаментальных фишек, которой часто бравируют сторонники криптовалют - [заполненная изначально дефляция](#). Это связано с тем, что еще на этапе проектировки системы, было указано суммарное ограничение в 21 миллион монет (примерно), и даже если очень сильно захотеть, поднять этот порог не получится. В отличие от рубля или доллара, которые по желанию казначейства могут

быть напечатаны в любом количестве, что иногда приводит к печальным последствиям, как в [Зимбабве](#).

BTW не все считают дефляцию таким уж [однозначным плюсом](#).

Reward

Следующий хороший вопрос - откуда взялась цифра в 21 миллион?

Я думаю вы понимаете, что **сумма выпущенных монет в конкретный момент времени равна сумме вознаграждений за блоки, созданные к этому моменту**. Довольно очевидный факт, учитывая что существует только один путь, по которому новые монеты попадают в сеть.

Но вознаграждение не фиксировано, и более того, каждые 210.000 блоков (примерно раз в 4 года) оно уменьшается в два раза.

```
consensus.nSubsidyHalvingInterval = 210000;
// https://github.com/bitcoin/bitcoin/blob/master/src/chainparams.cpp#L73
```

Так, например, когда все начиналось в январе 2009, награда за блок составляла 50 BTC. Спустя 210.000 блоков, в ноябре 2012 она упала до 25 BTC, и совсем недавно, 9 июля 2016, [снизилась до 12.5 BTC](#).

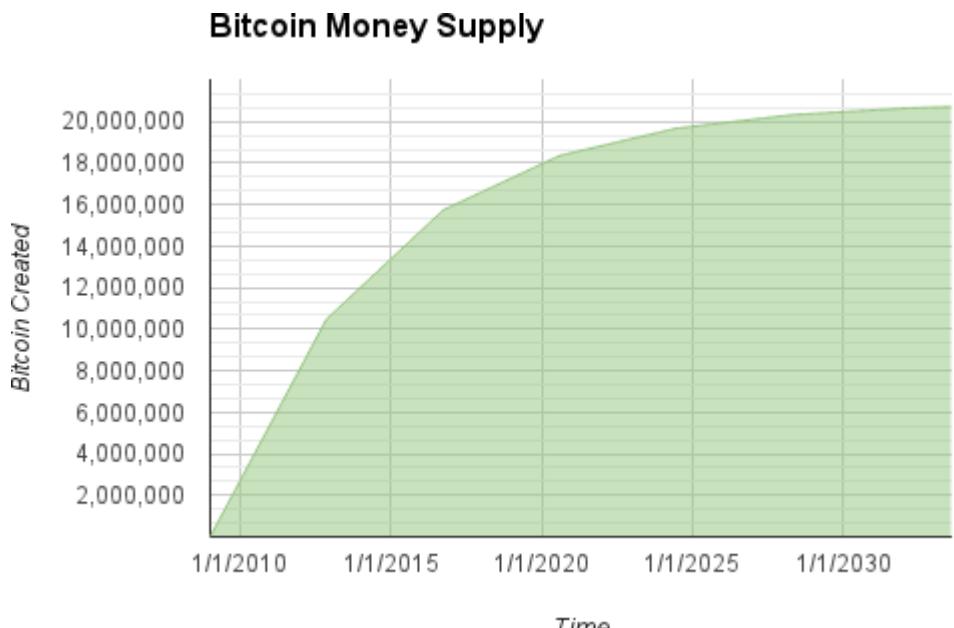
Несложно посчитать точное число Сатоши, которые будут произведены на свет, если предположить, что Bitcoin не заглохнет в ближайшие 200 лет:

```
start_block_reward = 50
reward_interval = 210000

def max_money():
    # 50 BTC = 50 0000 0000 Satoshis
    current_reward = 50 * 10**8
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total

print "Total BTC to ever be created:", max_money(), "Satoshis"
# Total BTC to ever be created: 209999997690000 Satoshi
```

На картинке ниже изображена кривая добычи, которая будет все более плавно подходить к отметке в 21 миллион BTC, достигнув пика примерно в 2140 году. В это время награда за блок станет 0 BTC.



Остается только гадать, что тогда произойдет с Bitcoin, но одно мы можем знать точно - совсем без денег майнеры не останутся. Как минимум у них еще есть *transaction fee*, другое дело, что эта самая комиссия может на порядок увеличиться.

Возьмем для наглядности какой-нибудь свежий блок, например [#447119](#). Сумма комиссий со всех транзакций в блоке составляет примерно 0.78 BTC, при том что вознаграждение за него - 12.5 BTC. То есть если завтра *reward* исчезнет, то в нашем случае комиссия должна вырасти более чем в 16 раз, чтобы нивелировать это неприятное событие. Понятно, что никакими микроплатежами тут уже и не пахнет.

Mining for dummies

Давайте постараемся еще раз представить процесс майнинга на нашем, пока что примитивном уровне.

Существует сеть с кучей участников. Некоторые из участников называют себя *майнерами* - они готовы собирать на своем ПК новые транзакции, проверять их на валидность, потом каким-то образом *майнить* из них новый блок, раскидывать этот блок по сети и получать за это денежку. Логичный вопрос - если все так просто, то почему этим не занимается каждый участник сети?

Понятно, что если все было бы так, как я сейчас описал, то блоки выходили бы по сто раз в секунду, валюты было бы столько, что за нее никто не дал бы и цента, и так далее.

Поэтому Сатоши был вынужден придумать алгоритм, со следующими свойствами:

- Создание нового блока - вычислительно сложная задача. Нельзя вот так просто включить мощный ПК и за минуту намайнить сто блоков.
- На вычисление нового блока всей сети уходит 10 минут (в среднем). Если посмотреть на Litecoin, то там блоки выходят раз в 2-3 минуты, суть заключается именно в том, что среднее время заранее установлено.
- Более того, это время не зависит от числа участников сети. Даже если однажды майнеров станет в сто раз больше, то алгоритм должен так изменить свои параметры, чтобы блок стало находить сложнее, и *block time* опустился обратно в окрестность десяти минут.

- Помним, что сеть распределенная и одноранговая, а значит, она должна сама понимать, в какой момент и как нужно подкрутить эти параметры. Никаких управляющих нод, все полностью автономно.
- Если решение задачи по созданию нового блока - это сложная задача, требующая времени и ресурсов, то проверка блока на "корректность" должна быть простой и практически мгновенной.

Proof-of-Work (PoW)

Скорее всего вы сейчас прибываете в полной прострации и не очень понимаете, как такое вообще возможно. Но Сатоши не растерялся и смог придумать решение для всех этих проблем - алгоритм получил название *Proof-of-Work*, вот так он выглядит (советую сначала прочитать [Bitcoin in a nutshell - Blockchain](#)):

Пусть вы - майнер. У вас есть 10 транзакций, которые вы хотите замайнить в блок. Вы проверяете эти транзакции на валидность, формируете из них блок, в поле *nonce* указываете 0 и считаете хэш блока. Потом меняете *nonce* на 1, снова считаете хэш. И так до бесконечности.

Ваша задача - найти такой *nonce*, при котором хэш блока (256 битное число) меньше заранее заданного числа N. Поиск такого хэша возможен только тупым перебором *nonce*, никаких красивых алгоритмов здесь нет. Поэтому чем быстрее вы хотите найти *nonce*, тем больше мощностей вам понадобится.

Число N - именно тот параметр (его еще называют *target*), который сеть настраивает в зависимости от суммарной мощности майнеров. Если завтра блоки начнут выходить, условно говоря, раз в три минуты, то N будет как-то уменьшено, времени на поиск *nonce* потребуется больше и *block time* снова вырастет до 10 минут. И наоборот.

Technical side



Самое время перейти от слов к делу и продемонстрировать, как работает *Proof-of-Work* и майнинг в целом. А по моему скромному мнению, нет ничего лучше, чем показать вообще весь процесс прямо в боевых условиях. Для этого мы сейчас с ходу напишем свою майнинг ноду и даже попробуем сделать новый блок раньше всех, хотя шансы на успех невелики.

Receive transactions

По-хорошему здесь нужно снова погружаться в спецификацию протокола, устанавливать контакт с другими нодами и ждать, пока нам пришлют свежие транзакции. В этом случае у нас получится самый настоящий real-time майнер, ничем не хуже уже готовых решений (но это не точно).

Я предлагаю пойти упрощенным путем. Открываем blockchain.info и выбираем несколько транзакций из списка "Последние транзакции". Они только-только попали в сеть и пока что не входят ни в один из блоков. Далее открываем другой block explorer - chainquery.com. Он умеет выдавать транзакции в сыром формате и по хэшам получаем транзакции в уже знакомом нам виде. Я ограничился двумя ([раз](#), [два](#)):

```
txn_pool = []
txn_pool.append("010000001440d795fa6267cbae00ae18e921a7b287eaa37d7f41b96ccbc61ef9a323a003d010000
006a47304402204137ef9ca79bcd8a953c0def89578838bbe882fe7814d6a7144eaa25ed156f66022043a4ab91a7ee3bf
58155d08e5f3f221a783f645daf9ac54fed519e18ca434aea012102965a03e05b2e2983c031b870c9f4afef1141bf30dc
5bb993197ee4a52f1443e0feffffff0200a3e111000000001976a914f1cfa585d096ea3c759940d7bacd8c7259bbd4d48
8ac4e513208000000001976a9146701f2540186d4135eec14dad6cb25bf757fc43088accbd50600")
txn_pool.append("010000001517063b3d932693635999b8daaed9ebf020c66c43abf504f3043850bca5a936d010000
006a47304402207473cda71b68a414a53e01dc340615958d0d79dd67196c4193a0ebcf0d9f70530220387934e7317b602
97f5c6e0ca4bf527faaad830aff45f1f5522e842595939e460121031d53a2c228aedcde79b6ccd2e8f5bcfb56e2046b46
81c4ea2173e3c3d7ffc686fffffff0220bcbe00000000001976a9148cc3704ccb6af566598fea13a3352b46f85958118
8acba2cfb09000000001976a914b59b9df3700adae0ea819738c89db3c2af4e47d188ac00000000")
```

Check

Следующим шагом нужно проверить полученные транзакции. Я этого делать не буду, просто перечислю основные пункты:

- Верно соблюдены структура и синтаксис транзакции
- Список входов / выходов не может быть пустым
- Транзакции на входе должны существовать либо в UTXO pool, либо в пуле неподтвержденных транзакций
- Сумма входов не меньше суммы выходов
- Полный список можете найти [здесь](#)

Некоторые майнеры отвергают транзакции с нулевой или слишком маленькой комиссией, но это каждый решает сам.

Sort

На всякий случай поясню, что [ничто не мешает вам](#) включать транзакции в блок в каком угодно порядке, главное, чтобы они прошли все проверки. В моем случае транзакций всего две, поэтому сортировать их тем более нет никакого смысла. Но не стоит забывать, что размер блока ограничен 1 МБ, поэтому если у вас в пуле 10.000 транзакций, то будет разумно отсортировать их по комисии и записать в блок только самые "дорогие".

BTW Часто попадаются статьи / книги, в которых сказано, что перед майнингом нового блока, Bitcoin Core сортирует транзакции по специальному параметру *priority*, который считается как

```
Priority = Sum (Value of input * Input Age) / Transaction Size
```

Это было верно вплоть до версии 0.12.0, потом сортировку по *priority* [отключили](#).

Get reward

version	02000000	
previous block hash (reversed)	17975b97c18ed1f7e255adf297599b55 330edab87803c817010000000000000000	Block hash 0000000000000000 e067a478024addfe cdc93628978aa52d 91fabd4292982a50
Merkle root (reversed)	8a97295a2747b4f1a0b3948df3990344 c0e19fa6b2b92b3a19c8e6badc141787	
timestamp	358b0553	
bits	535f0119	
nonce	48750833	
transaction count	63	
coinbase transaction		
transaction		
...		

Если вы посмотрите на структуру любого блока, то самой первой всегда идет так называемая *coinbase* транзакция - именно она отправляет вознаграждение на адрес майнера. В отличии от обычных транзакций, *coinbase transaction* не тратит в качестве входов выходы из *UTXO pool*. Вместо этого у нее указан только один вход, называемый *coinbase*, который "создает" монеты из ничего. Выход у такой транзакции тоже только один. Он отправляется на адрес майнера награду за блок плюс сумму комиссий со всех транзакций в блоке. В моем случае это $12.5 + 0.00018939 + 0.0001469 = 12.50033629$.

Давайте подробнее рассмотрим структуру *coinbase* транзакции, а если конкретнее - ее вход. На всякий случай напомню, как выглядит вход у "обычной" транзакции:

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent, first one is 0
1-9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script.
4 bytes	Sequence Number	Currently disabled Tx-replacement feature, set to 0xFFFFFFFF

Вот три отличия входа *coinbase* транзакции:

- Вместо настоящего *transaction hash* указывается 32 нулевых байта
 - Вместо *output index* указывается `0xFFFFFFFF`.
 - В поле *unlocking script* можно указать что угодно размером от 2 до 100 байт, поэтому это поле еще называют *coinbase data*. Например в *genesis block* там спрятана фраза `"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"`. Как правило, майнеры вставляют в *coinbase data* свое имя / имя майнинг пула / еще что-нибудь.

Часто в *coinbase data* вставляют так называемый *extra nonce*, подробнее [здесь](#). Суть в том, что может не найтись нужного *nonce*, при котором хэш блока меньше *target* (на самом деле это будет происходить в большинстве случаев). Тогда остается что-нибудь менять в транзакции, чтобы получились другие хэши, например - *UNIX timestamp*. Но если вы читали [Bitcoin in a nutshell - Blockchain](#), то знаете, что *timestamp* тоже сильно не изменишь, иначе другие ноды отвергнут ваш блок. Решение оказалось довольно простым: достаточно добавить какое-нибудь число в *coinbase data* и менять его, если для текущего *header* не нашлось нужного *nonce*.

Процесс создания новой транзакции подробно описан в главе [Bitcoin in a nutshell - Protocol](#), поэтому здесь я просто приведу уже полученную *coinbase transaction*, весь код, как обычно, доступен на [Github](#):

Осталось только посчитать для этих трех транзакций *merkle root*. Для этого воспользуемся фрагментом кода из [Bitcoin in a nutshell - Blockchain](#):

```
txn_pool.insert(0, coinbase_txn)
txn_hashes = map(getTxnHash, txn_pool)

print "Merkle root: ", merkle(txn_hashes)
# Merkle root: 4b9ff9ab901df82050f858accde99b9169067aca0aeade25598ea5505fb53836
```

Target

Как я уже написал выше, весь майнинг сводится к тому, чтобы найти хэш блока меньше числа, называемого *target*. В структуре блока это число записывается в поле *bits*, например для блока #277,316, *target* равнялся 1903a30c.

В bits на самом деле записаны сразу два числа: первый байт `0x19` - экспонента, оставшиеся три байта `0x03a30c` - мантисса. Для того, чтобы получить target из bits, нужно воспользоваться следующей формулой: `target = mantissa * 2^(8 * exponent - 3)`. В случае блока #277.316 получается:

BTW чем меньше target, тем больше difficulty и наоборот.

PoW

Теперь, когда вы разобрались со всеми нюансами, можно запускать майнер:

```

import hashlib
import struct
import sys

# ===== Header =====
ver = 2
prev_block = "0000000000000000e5fb3654e0ae9a2b7d7390e37ee0a7c818ca09fde435f0"
merkle_root = "6f3ef687979a1f4866cd8842dcbebd2e47171e54d1cc76c540faecafe133c39"
bits = 0x10004379 # Not the actual bits, I don't have synced blockchain
timestamp = 0x58777e25
# Calculate current time with this code:
# hex(int(time.mktime(time.strptime('2017-01-12 13:01:25', '%Y-%m-%d %H:%M:%S')))) -
# time.timezone)

exp = bits >> 24
mant = bits & 0xfffffff
target_hexstr = '%064x' % (mant * (1 << (8 * (exp - 3))))
# '0000000000000000000000000000000043790000000000000000000000000000'
target_str = target_hexstr.decode('hex')
# ===== Header =====

nonce = 0
while nonce < 0x10000000: # 2**32
    header = ( struct.pack("<L", ver) + prev_block.decode('hex')[::-1] +
               merkle_root.decode('hex')[::-1] + struct.pack("<LLL", timestamp, bits, nonce))
    hash = hashlib.sha256(hashlib.sha256(header).digest()).digest()

    sys.stdout.write("\rNonce: {}, hash: {}".format(nonce, hash[::-1].encode('hex')))
    sys.stdout.flush()

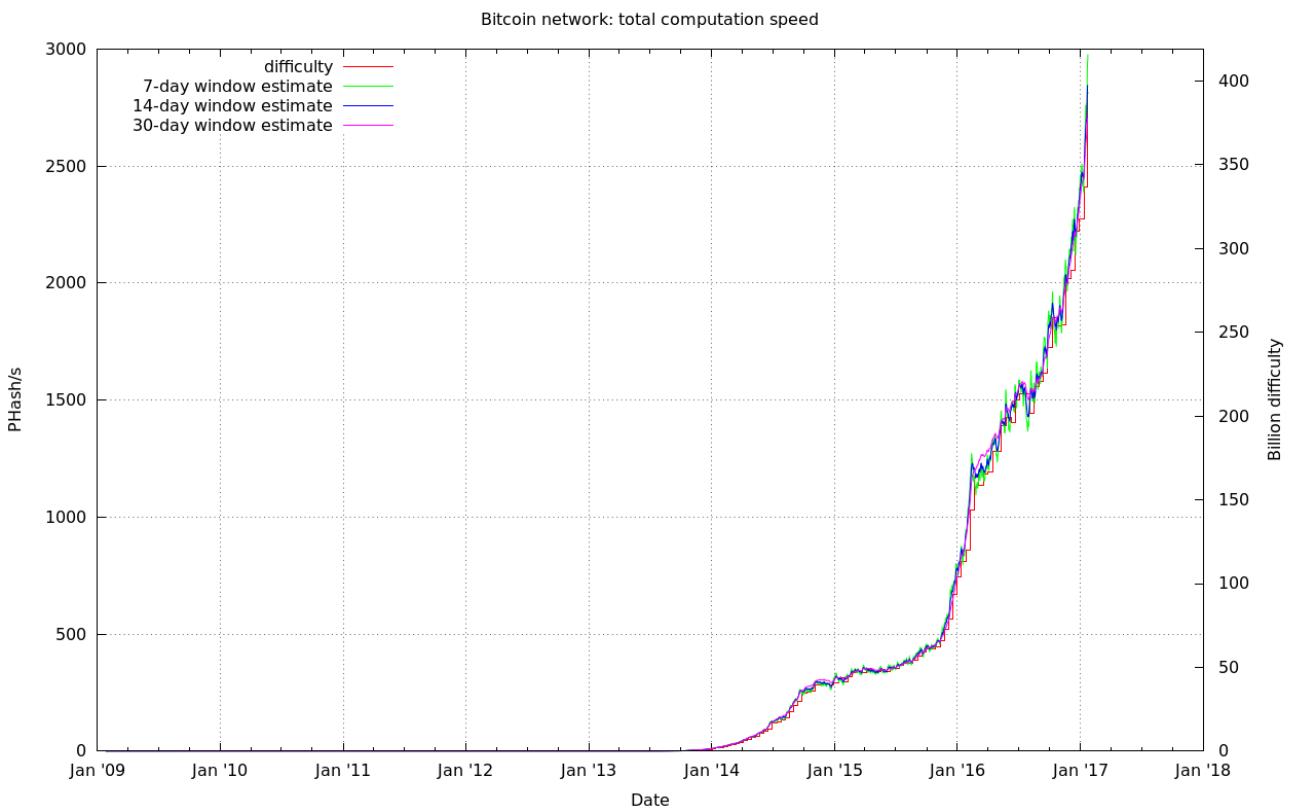
    if hash[::-1] < target_str:
        print 'Success!'
        break
    nonce += 1

```

Hash rate

Если вы дождались заветной строчки `Success!`, то у вас либо Intel Core i7, либо очень много свободного времени. Я понятия не имею, когда этот код закончит свою работу и найдет ли он *nonce* вообще, потому что текущая сложность просто чудовищно велика. Даже если предположить, что наша программа способна обсчитать 100.000 хэшей в секунду (а это не так), то она все равно в миллионы раз медленней любого настоящего майнера, поэтому на поиск *nonce* у нее может уйти несколько дней.

Чтобы вы осознали масштаб проблемы: существует метрика [hashrate](#). Она отражает суммарную мощность майнеров в сети Bitcoin, единица измерения - хэши SHA256 в секунду. Вот ее [график](#):



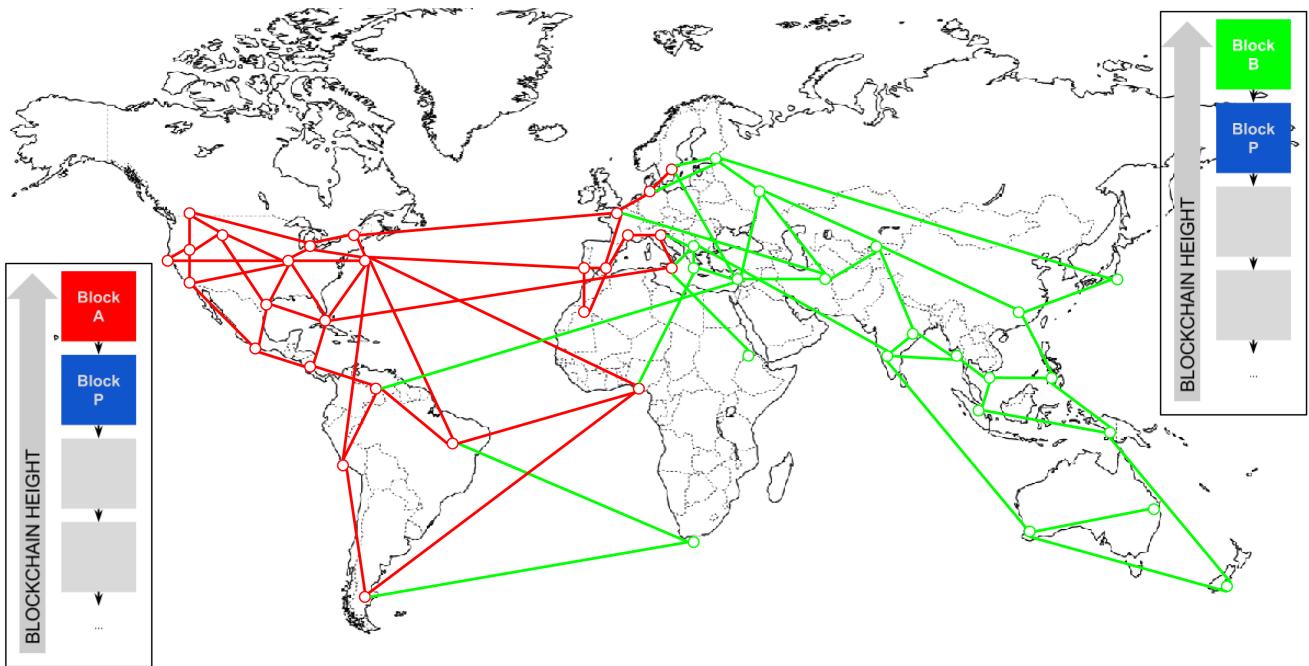
Будем считать, что хэшрейт составляет $2.000 \text{ PH/s} = 2.000.000 \text{ TH/s} = 2.000.000.000 \text{ GH/s} = 2.000.000.000.000 \text{ MH/s} = 2.000.000.000.000.000 \text{ KH/s}$. А наша программа даже 100 KH/s не может осилить, поэтому соревноваться со всей сетью нет никакого смысла.

2 Blocks 1 Chain

Fork

Давайте на минуту представим, что майнеры ищут блок #123456. И примерно в одно и то же время он был найден двумя независимыми майнерами, один из которых живет в Австралии, а другой в США. Каждый из них начинает раскидывать свою версию блока по сети, и в результате получается, что у одной половины мира один блокчейн, а у другой - другой.

Возможно ли такое и что произойдет в этом случае?



Да, возможно. Более того, такое происходит довольно часто. В этом случае каждая нода продолжает придерживаться своей версии блокчейна до тех пор, пока кто-нибудь не найдет следующий блок. Предположим, что новый блок продолжает "зеленую" ветку, как на картинке ниже.



В этом случае те ноды, которые придерживаются "красной" версии, автоматически синхронизируют зеленую, потому что в мире Bitcoin работает правило: **"истинна" самая длинная версия блокчейна.** "Красная" версия блокчейна будет попросту забыта, вместе с наградами для тех, кто ее нашел.

Вы можете спросить: а что если форк пойдет дальше? То есть одновременно с "фиолетовым" блоком найдут еще один, который будет продолжать "красную" версию блокчейна?

Скорее всего, эту книгу будут читать не только люди с хорошим математическим образованием, поэтому дам самый общий ответ - такое безусловно возможно. Но вероятность даже первого форка довольно мала, второго - еще меньше и так далее. Чтобы вы понимали, самый длинный форк за всю историю Bitcoin составил всего [4 блока](#). Так что в какой-то момент одна из веток все таки вырвется вперед, и вся сеть перейдет на нее.

Если вам интересна эта проблема именно с ракурса теории вероятностей, то можете прочесть "[What is the probability of forking in blockchain?](#)" Еще этот вопрос неплохо расписан в знаменитой "[Bitcoin: A Peer-to-Peer Electronic Cash System](#)" by Satoshi Nakamoto.

51% attack

На том простом факте, что в блокчейне самая длинная цепочка - доминирующая, основана целая атака:

Представьте, что вы мошенник и покупаете товар на 1000 BTC в каком-нибудь магазине. Вы договариваетесь с продавцом и отправляете ему деньги. Продавец проверяет блокчейн, видит, что такая транзакция действительно была, прошла все проверки и даже попала в какой-нибудь блок, например #123. После этого продавец идет на почту и отправляет вам товар.

В это время вы включаете свою майнинг-ферму и начинаете майнить, **начиная с блока #122**. Если у вас достаточно мощностей, то вы можете обогнать всю остальную сеть и быстрее всех досчитать до блока #124, после чего весь мир перейдет на вашу версию блокчейна. При этом свою транзакцию на 1000 BTC, вы не будете включать ни в один из блоков, а значит она будет навсегда забыта, как будто ее никогда и не было. В результате продавец лишится товара и не получит своих денег.

Не буду вдаваться в теорию вероятностей, но осуществить такую атаку невозможно, если только у вас нет как минимум половины хэшрейта всей сети. Подробнее можете прочитать в [bitcoin.pdf](#).

Тем не менее некоторые майнинг пулы обладают очень значительными мощностями. Так например BTC Guild в 2013 году [почти преодолел](#) порог в 51% хэшрейта. В какой-то момент они замайнили сразу 6 блоков подряд, так что при желании смогли бы осуществить данную атаку. Поэтому рекомендуется считать транзакцию *подтвержденной* только после того, как было создано 6 блоков сверху.

Hardware

Можете сразу забыть про майнинг на CPU или GPU. Чтобы вы понимали, ниже изображен [хэшрейт](#) на начало 2017 года. Будем считать, что он в среднем составляет 2.300.000 TH/S, то есть 2.300.000.000 MH/s. Для сравнения, самые зверские видеокарты, такие как [ATI Radeon™ HD 5870 Eyefinity](#) или [AMD Radeon HD 7970 \(x3\)](#), выдают [в лучшем случае 2000 MH/S](#). Среди процессоров первое место занимает [Xeon Phi 5100](#) со смешными 140 MH/s.



Так что даже исходя из курса в 1000 \$/BTC и имея на руках 10.000 MH/s, вы в среднем будете зарабатывать [20 центов в месяц](#).

Difficulty Factor	3.36899932796e+11	
Hash Rate	10000.0	MH/s ▾
Exchange Rate	1000.0	(\$/BTC) [user]
BTC / Block	12.50000000	
Calculate		

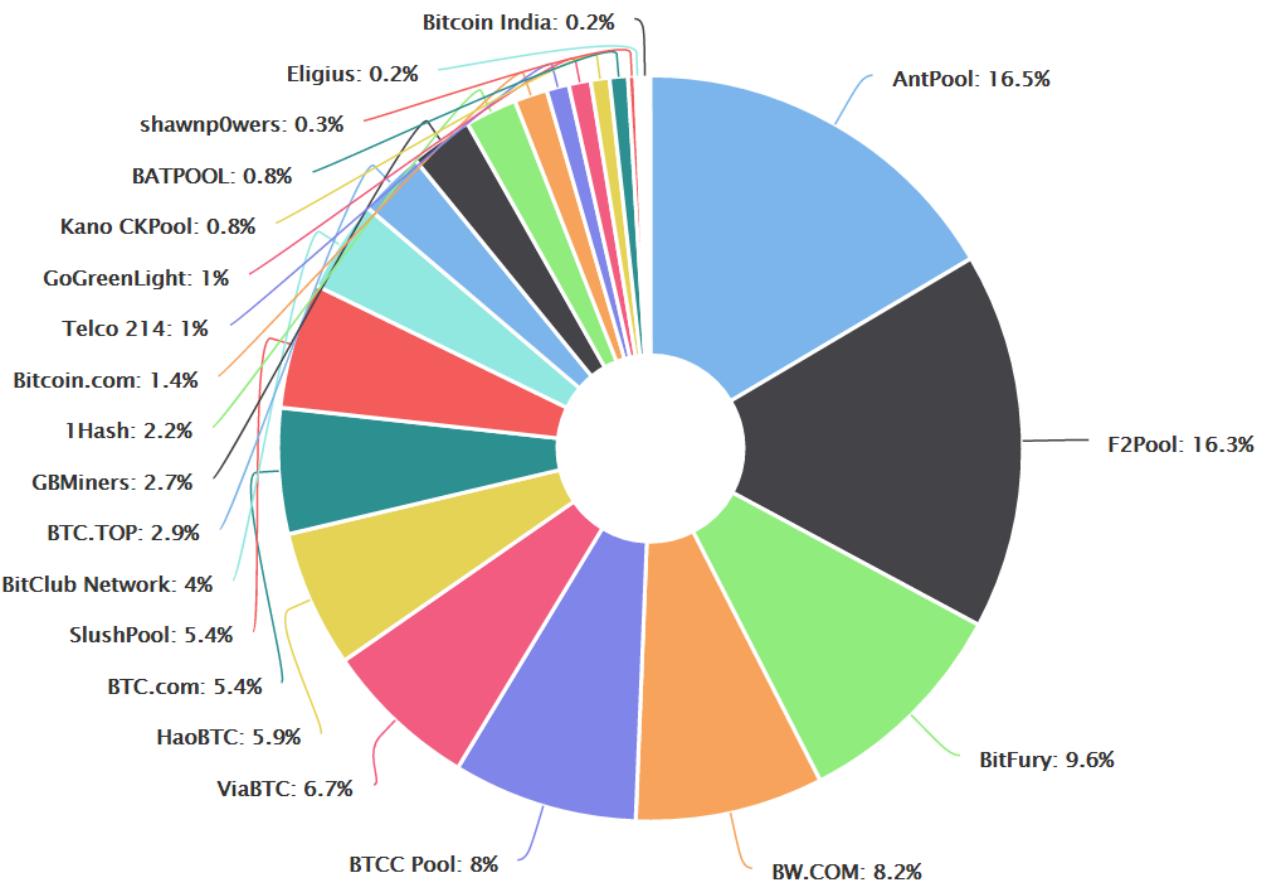
This Difficulty			Next Difficulty [estimated]		
	Coins	Dollars		Coins	Dollars
per Day	0.00000746 BTC	\$0.01	per Day	0.00000626 BTC	\$0.01
per Week	0.00005225 BTC	\$0.05	per Week	0.00004382 BTC	\$0.04
per Month	0.00022718 BTC	\$0.23	per Month	0.00019052 BTC	\$0.19
this diff (est)	0.00006920 BTC	\$0.07			

CPU майнинг перестал быть рентабельным еще в 2011 году, GPU держался примерно до 2013 года, но тоже прогорел, когда широкое распространение получили так называемые *application-specific integrated circuit* - [ASIC](#). Это специальные чипы, заточенные под майнинг на уровне железа. Самые простые стоят в районе 100\$, что сильно дешевле топовой видеокарты, но при этом способны выдавать от 1 TH/s.

То есть при прочих равных, имея два [Antminer S9](#) по 3.000\$ за штуку, вы будете зарабатывать почти [700 долларов в месяц](#) (без учета счетов за электричество)

Miner	Hash Power	Price
	Antminer S5	1.16 TH/s \$139.99
	Antminer S7	4.73 TH/s \$489.99
	Antminer S9	14.0 TH/s \$3,000
	Avalon 6	3.50 TH/s \$559.95
	SP20 Jackson	1.3-1.7 TH/s \$90.00

Но и на этом еще не все. Вы можете объединиться с другими майнерами в *mining pool* и начать майнить вместе, а заработанные деньги делить пропорционально вложенным мощностям. Это, очевидно, намного выгодней, чем пытаться заработать хоть что-нибудь в одиночку, поэтому именно пулы на сегодняшний день составляют главную движущую силу в мире майнинга. На начало 2017 года [основными игроками](#) на рынке пулов являются [AntPool](#), [F2Pool](#) и [Bitfury](#), обеспечивающие более 40% хэшрейта всей сети.



Conclusion

На этой высокой ноте я заканчиваю свой рассказ про техническое устройство Bitcoin. Исходники текста плюс примеры кода [здесь](#), там же pdf версия. Pull requests welcome, задавайте свои вопросы в Issues или в комментариях.



Links

-
- [Bitcoin mining the hard way: the algorithms, protocols, and bytes](#)
 - [Mining Bitcoin with pencil and paper: 0.67 hashes per day](#)
 - [Mining hardware comparison](#)
 - [Bitcoin: история развития, ASIC](#)
 - [Biggest & Best Bitcoin Mining Pools and Companies](#)