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.

Table of content

Introduction

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

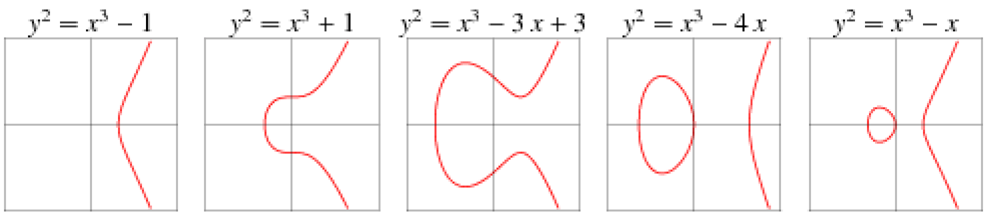
В Bitcoin используется так называемая криптография на эллиптических кривых (*Elliptic curve cryptography, ECC*). Она основана на некоторой особой функции - эллиптической кривой (не путать с эллипсом). Что это за функция и чем она так примечательна я расскажу дальше.

Elliptic curve

> Эллипти́ческая крива́я 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:

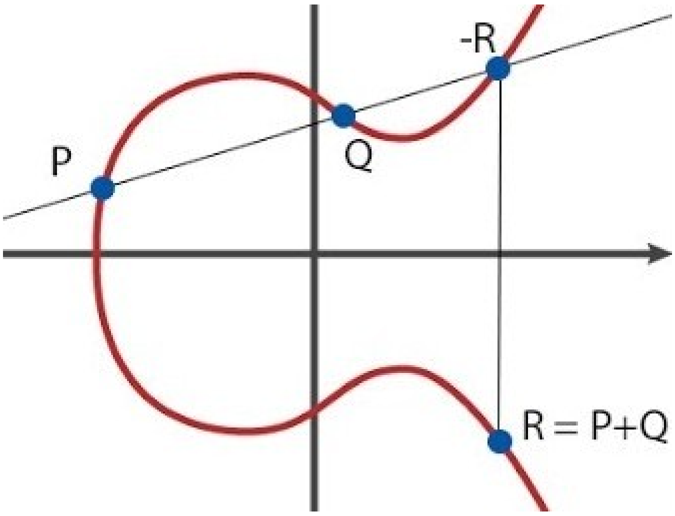


Скрипт для отрисовки графика на Python:

```
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()
```

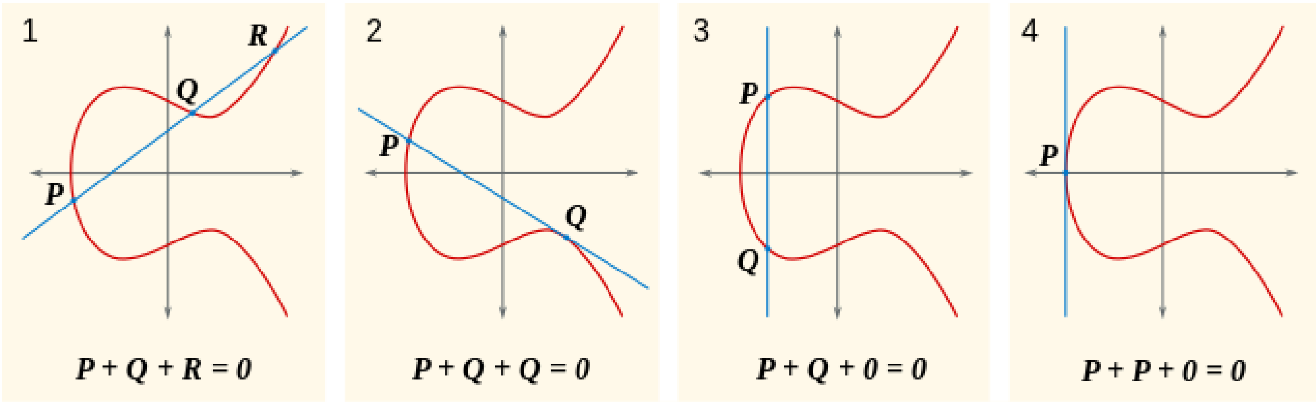
Если верить вики, то впервые эта функция засветилась еще в трудах Диофанта, а позже, в 17 веке, ей заинтересовался сам Ньютон. Его исследования во многом привели к формулам сложения точек на эллиптической кривой, с которыми мы сейчас познакомимся. Здесь и в дальнейшем мы будем рассматривать некоторую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

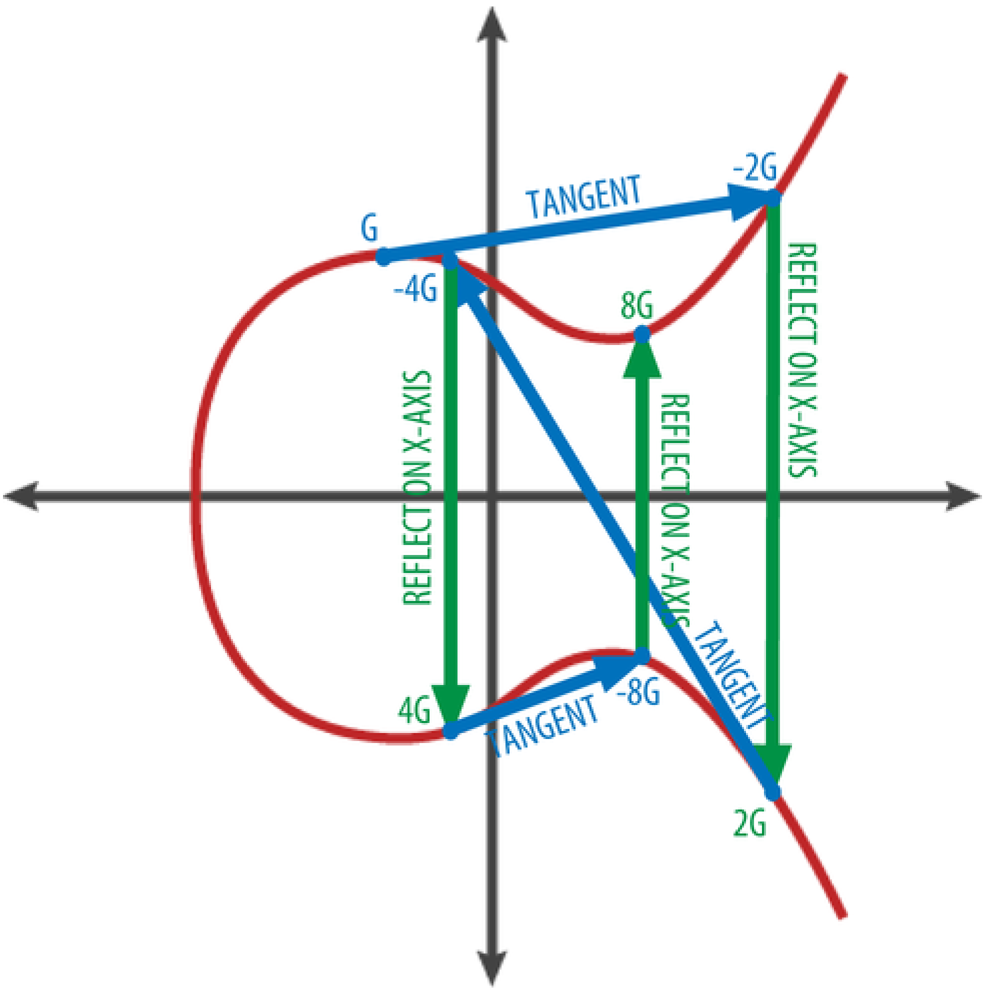
Самые сообразительные уже задались вопросом - а что будет, если например провести прямую через две точки, имеющие координаты вида $P(a, b)$ и $Q(a, -b)$, то есть прямая, проходящая через них, будет параллельна оси ординат (третий кадр на картинке ниже).



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.

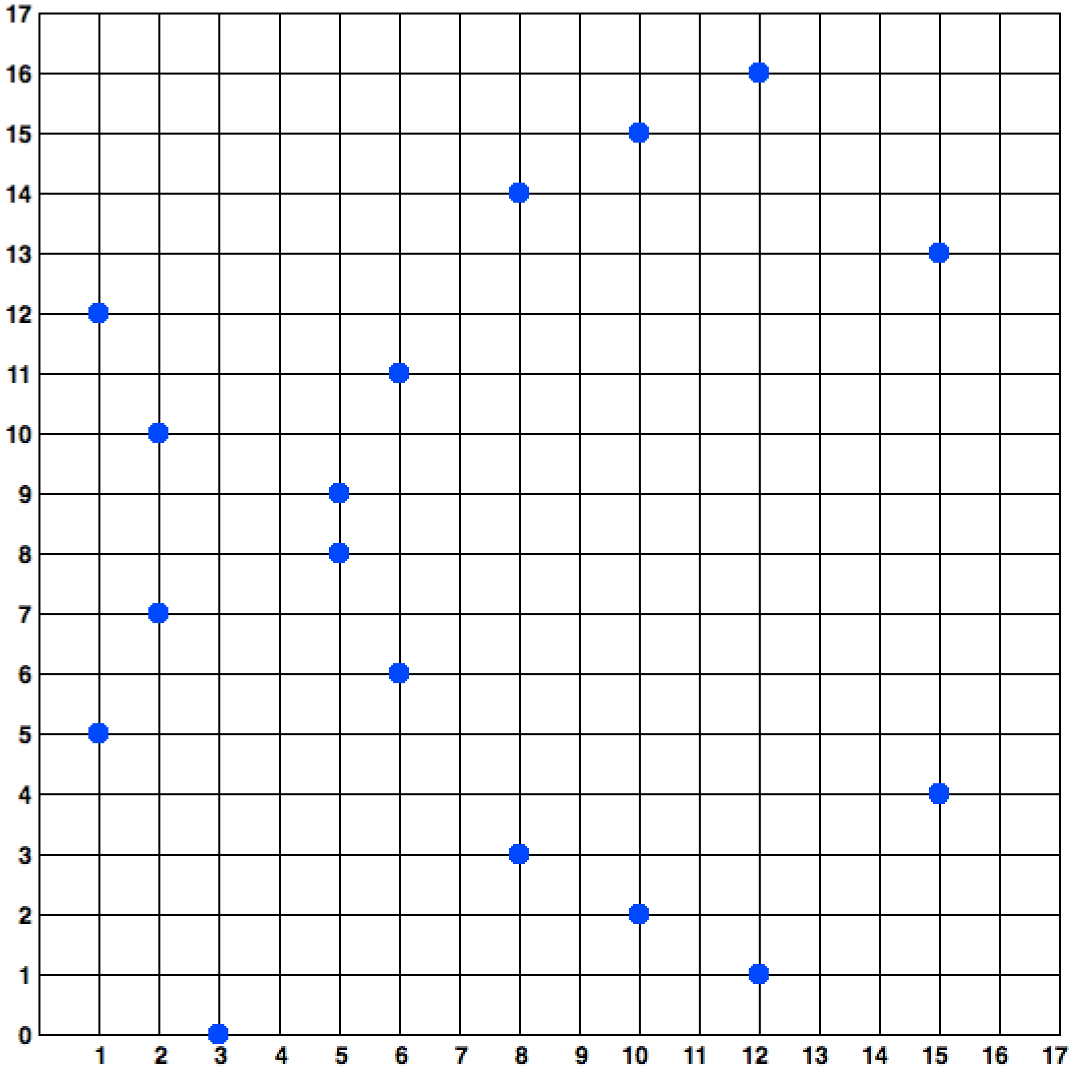
Теперь, легким движением руки, можно ввести операцию умножения точки на какое-то $\mathbb{N}$ число. В результате получим новую точку $K = G * k$, то есть $K = G + G + ... + G,\ k$ раз. С картинкой все должно стать вообще понятно:



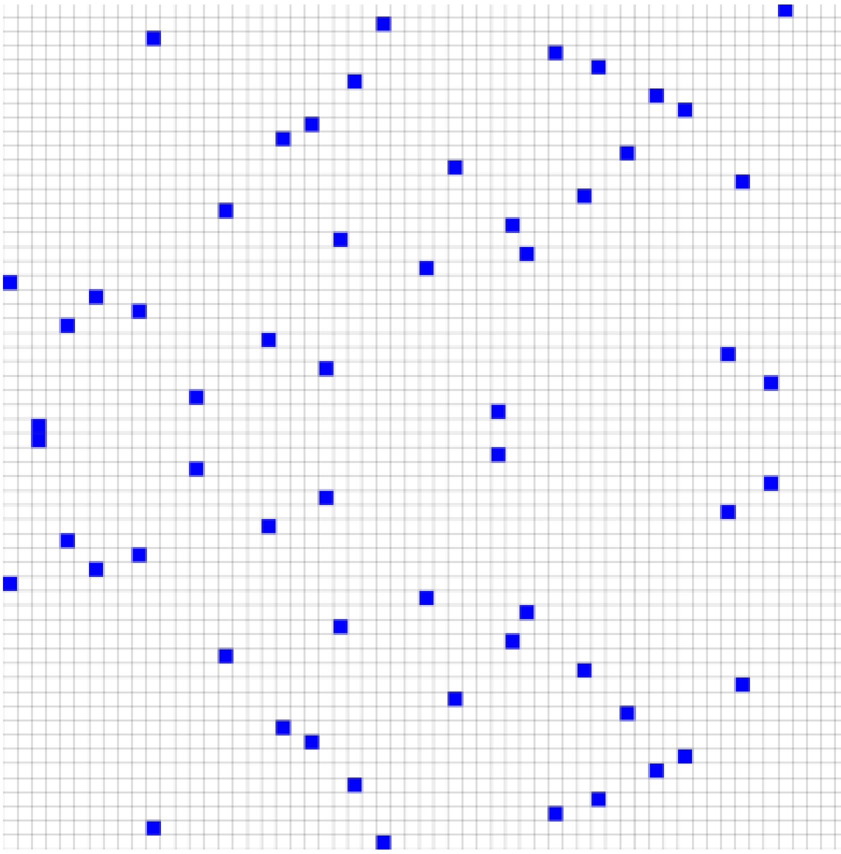
Elliptic curve over a finite field

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

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

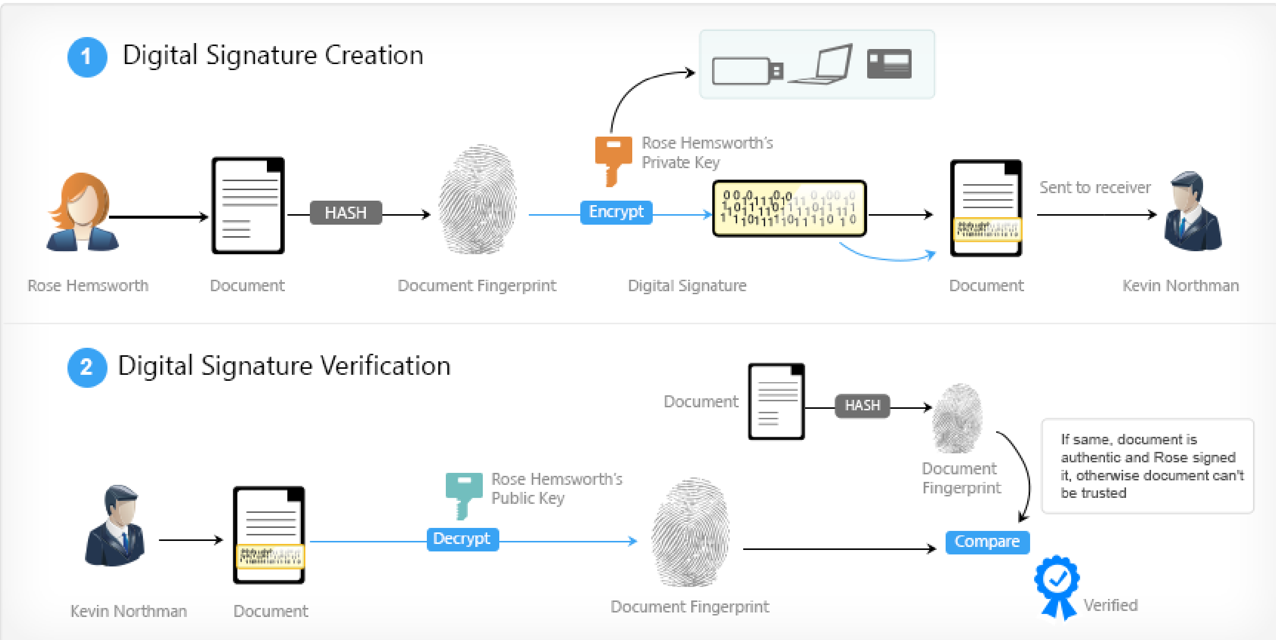
> Электро́нная по́дпись (ЭП), Электро́нная цифровая по́дпись (ЭЦП) — реквизит электронного документа, полученный в результате криптографического преобразования информации с использованием закрытого ключа подписи и позволяющий проверить отсутствие искажения информации в электронном документе с момента 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 рассылает 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

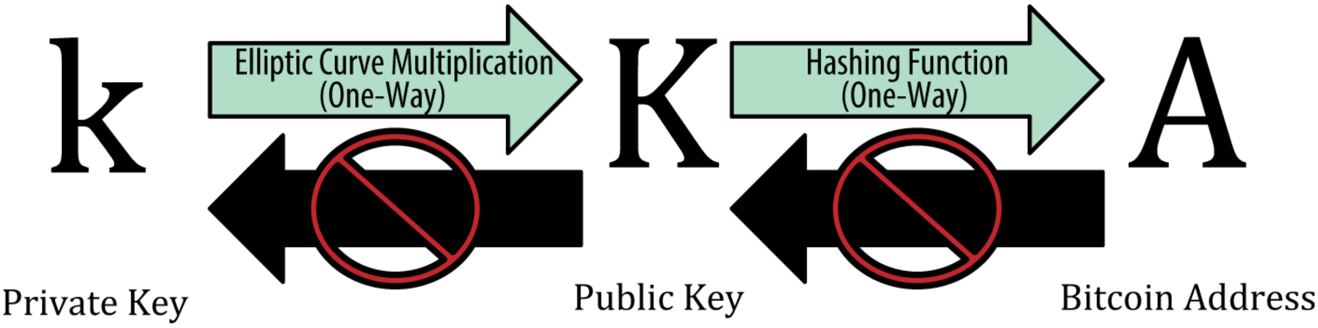
```
$ bitcoin-cli getnewaddress
14RVpC4su4PzSafjCKVWP2YBHv3f6zNf6U
$ bitcoin-cli dumpprivkey 14RVpC4su4PzSafjCKVWP2YBHv3f6zNf6U
L3SPdkFWMnFyDGyV3vkCjroGi4zfD59Wsc5CHdB1LirjN6s2vii9 // Format is different, explanation bellow
```

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'D5C08F1BFC9C26A5D18FE9254E7923DEBBD34AFB92AC23ABFC6388D2659446C1F04CCDEBB677EAABFED9294663EE79D
71B57CA6A6B76BC47E6F8670FE759D746'
```

C++, libbitcoin

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

int main() {
  // Private key
  bc::ec_secret secret = bc::decode_hash(
  "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
  // 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 0OIl 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.
> ```

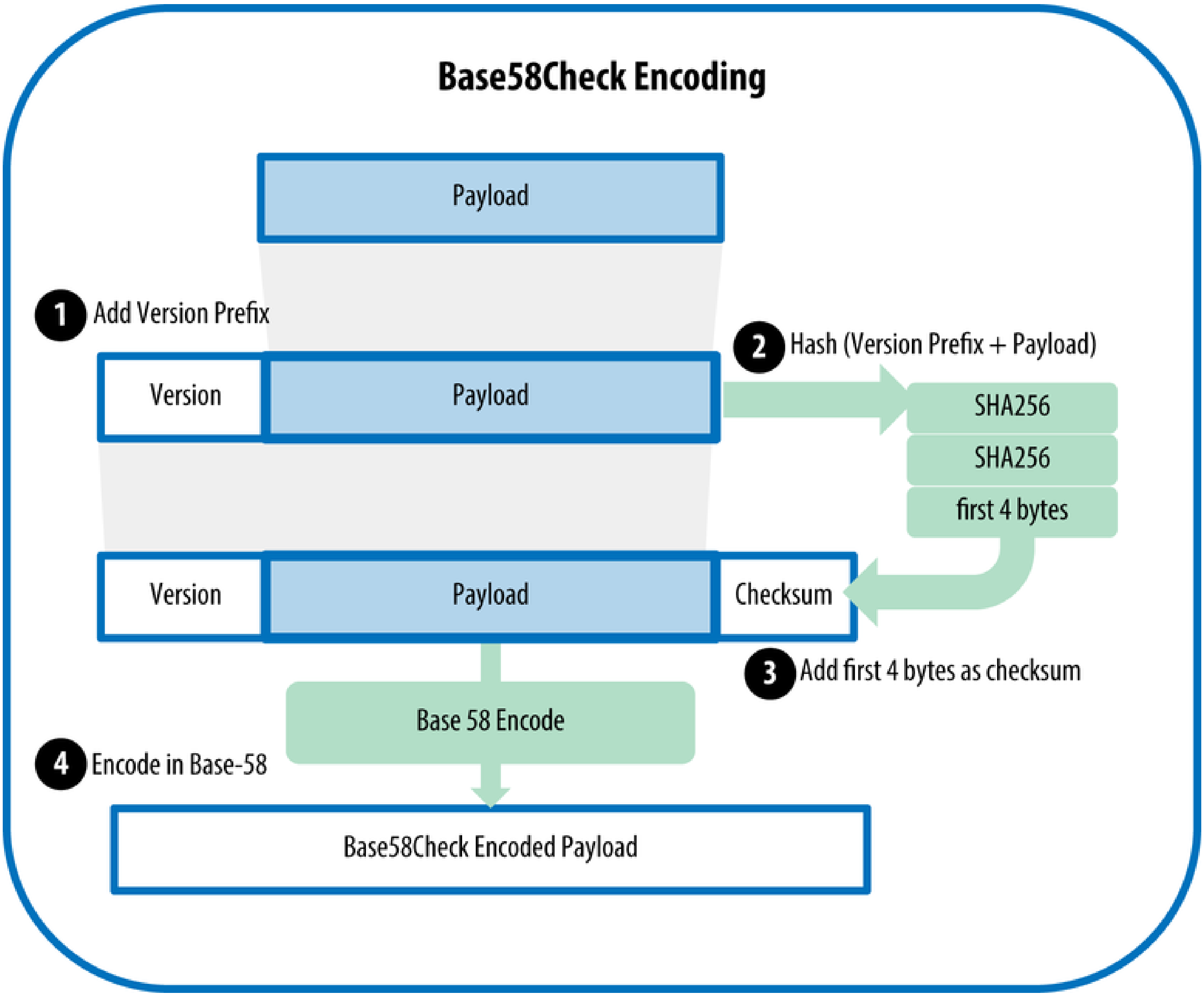
Краткость записи проще всего реализовать, используя довольно распространенную кодировку Base64, то есть используя систему счисления с основанием 64, где для записи используются цифры `0,1,...,9`, буквы `a-z` и `A-Z` - это дает 62 символа, оставшиеся два могут быть чем угодно, в зависимости от реализации.

Первое отличие Base58Check в том, что убраны символы `0,O,l,I` на случай, если кто-нибудь решит их перепутать. Получается 58 символов, можете проверить

```
b58 = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'

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 = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
    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))
```



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 ('\04' + 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
```

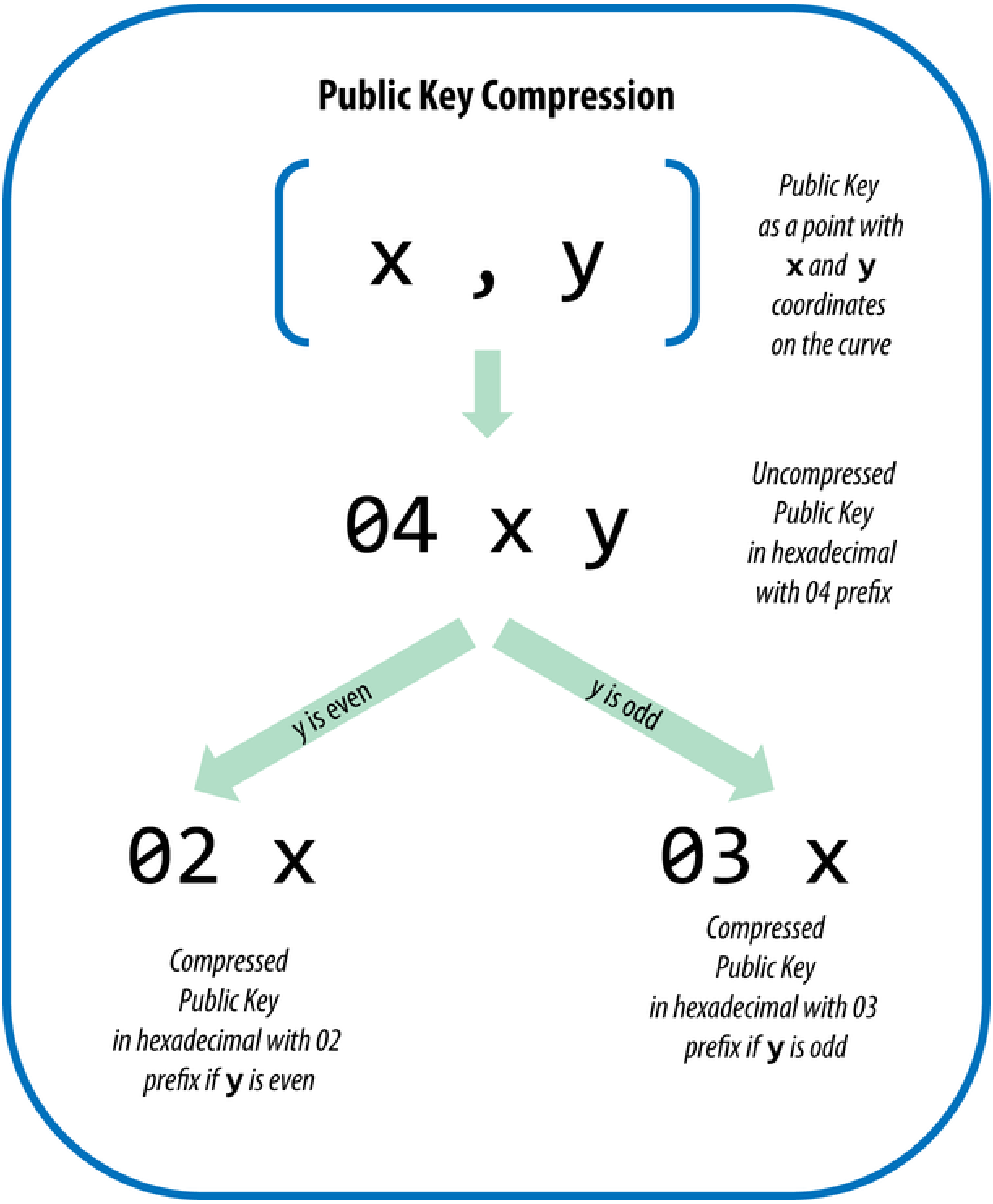
Однако, как можно догадаться из названия, это не самый оптимальный способ хранить публичный ключ.

Вы удивитесь, но второй формат называется *compressed*. Суть его в следующем: публичный ключ - это точка на кривой, то есть пара чисел удовлетворяющая уравнению $y^2 \; mod \; p = x^2 + ax + b \; (mod \; p)$. 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 `04162ebcd38c90b56fbdb4b0390695afb471c944a6003cb334bbf030a89c42b584f089012beb4842483692bdff9fcab8676fed42c47bffb081001209079bbcb8db`.
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