

The Bitcoin protocol

Balázs Kőműves

Falkstenen AB

Budapest, 30 May 2014

Overview of the talk

I will try to explain how Bitcoin works under the hood, going into details as far as time and my limited knowledge allows. I will cover the Bitcoin protocol as of mid-2013 (so newer developments like SPV, HD wallets or the payment protocol will be excluded).

Short overview:

- ▶ overview of Bitcoin
- ▶ the blockchain
- ▶ structure of a block
- ▶ mining
- ▶ bitcoin addresses
- ▶ transactions
- ▶ scripts
- ▶ elliptic curves
- ▶ ECDSA

High-level overview of Bitcoin

Bitcoin is a set of protocols and software which allows individuals and organizations:

- ▶ to send digital currency tokens to each other over the internet
- ▶ almost instantaneously (depending on confidence level, somewhere between 1 minute and 1 hour)
- ▶ cheaply (approximately 1-10 HUF per transaction at the moment, independently of the value of the tokens)
- ▶ pseudo-anonymously (more on this later)
- ▶ without the ability to cheat (uses digital signatures for proof of ownership)
- ▶ without any centralized control (distributed protocol)
- ▶ with a prefixed supply curve of said tokens

Low-level overview of Bitcoin

Bitcoin is a decentralized, peer-to-peer digital currency protocol.

- ▶ bitcoin *addresses* correspond to private/public key pairs
(more precisely, addresses are hashes of public keys)
- ▶ addresses can hold amounts of *coins* (varying in time)
- ▶ transactions *spend* the output(s) of previous transaction(s)
- ▶ a transaction specifies how its output(s) can be spend in the future
 - ▶ typical transaction: send the coins to one or more address(es)
 - ▶ the spending condition: the next spender(s) must prove that they own the private key(s) corresponding to these addresses
- ▶ transactions are stored in the *public ledger* (which is a decentralized database)
- ▶ it's the the set of transactions which is fundamental - the "coins" are just invariants.

Public key cryptography

Each party has a *pair* (d, Q) of corresponding keys: Q is public (say, on published on their homepage) and d is private (typically only a single person knows it).

Main applications:

- ▶ Establishing a shared secret without meeting (key exchange)
- ▶ Send messages which only the intended recipient can decrypt (encryption)
- ▶ Prove that a message was really written by the person who claims it (signature)

From these basic building blocks, a huge set of really interesting applications can be built. Bitcoin only uses the signature algorithm.

Public-key crypto is widely used on the internet today: HTTPS, SSH, TLS, PGP, Bitcoin, OTR messaging, etc...

Bitcoin transaction example, part I.

I expect income from two sources:

- ▶ I sold a book to a Alice for 0.3 BTC
- ▶ I made a bet with Bob, and won 0.2 BTC

I generate two key-pairs (d_1, Q_1) and (d_2, Q_2) and derive the corresponding addresses A_1 and A_2 . Give the address A_1 to Alice and the address A_2 to Bob. They send me the agreed amounts. Two transactions T_1 and T_2 will appear in the ledger:

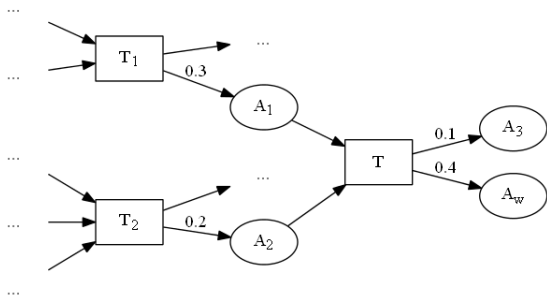
- ▶ T_1 says that 0.3 BTC can be spent by anybody who can prove that they know d_1
- ▶ T_2 says that 0.2 BTC can be spent by anybody who can prove that they know d_2

This means that I “own” 0.5 BTC altogether (since nobody else knows d_1 or d_2).

Next, let's say I want to buy a bottle of whisky for 0.4 BTC!

Bitcoin transaction example, part II.

To send 0.4 BTC for the address of the whisky shop A_w , I need to craft a transaction T which “spends” the outputs of the previous transactions T_1 and T_2 , sends 0.4 BTC of it to the whisky shop, and sends the remaining 0.1 BTC back to me (usually to a freshly generated address A_3)



The transaction T is signed with the private keys corresponding to the addresses A_1 and A_2 , to prove that they belong to me.

The blockchain

The public ledger, called *the blockchain* in Bitcoin-slang, is a distributed, (mostly) append-only database.

- ▶ the blockchain is built up from *blocks*; on average 1 new block is produced and appended every 10 minutes
- ▶ each block contains a small header, and a set of transactions
- ▶ the blocks are organized in a linked list (chain), starting from the “genesis block”
- ▶ each block is produced by a single participant “node”; to decide which node will produce the next block, the protocol uses a hash based lottery, with winning chance proportional to CPU-power (this process is usually called “mining”, which is rather confusing)
- ▶ people wanting to transact simply broadcast their (signed) transactions to the nodes
- ▶ to incentivise nodes to participate in this process, a reward is given to the winner, and they also get all the transaction costs for the set of transactions they include in that block
- ▶ typically, each participant node has a full copy of the database (currently around 18 gigabytes)

A block

A block consists of a block header (80 bytes), plus a set of transactions.

The header consists of:

- ▶ block version number (4 bytes; currently `0x00000002`, little-endian)
- ▶ hash of the previous block's header (32 bytes)
- ▶ the *Merkle root* of all transaction in this block (32 bytes)
- ▶ a unix timestamp (4 bytes)
- ▶ the current *difficulty target* (4 bytes)
- ▶ a *nonce* (4 bytes)
- ▶ (number of transactions (varint) - not actually part of the header)

A block is considered valid, if:

- ▶ all the previous blocks in the chain are valid
- ▶ the hash of the *block header* is smaller than the current target
- ▶ all the individual transactions are valid, and the Merkle root matches

Merkle trees

A *Merkle tree* is a hash tree. Bitcoin uses double-SHA256 as its primary hash function, that is, $\text{hash} = \text{SHA256}(\text{SHA256}(\text{msg}))$.

- ▶ for each transaction, a transaction hash is computed
- ▶ a binary tree is built, such that the leaves are labeled with the hashes of the individual transactions
- ▶ rows with odd number of elements are extended to be even by duplicating the last hash
- ▶ for each node, the hashes of its two children are concatenated, and the node is labelled by the hash of this string
- ▶ the Merkle root is the hash at the root node

If the set of transactions changes any way, the Merkle root also changes (and it changes unpredictably).

The lottery (or “mining”, or proof-of-work)

To be able to decide who produces the next block, a lottery process is used (confusingly called “mining”):

- ▶ the Merkle tree of the set of current transactions is computed
 - ▶ the first transaction must be always the so-called *coinbase transaction* (or *generation*), which gives a reward to the miner
 - ▶ thus the Merkle root will be different for all participants
 - ▶ these days, an extra nonce is also contained in this transaction
- ▶ the *nonce* (and the extra-nonce) is set to whatever (typically increasing numbers for simplicity)
- ▶ the hash of the block header is computed
- ▶ if the hash is smaller than the *current target*, we won, and broadcast the new block to the network
- ▶ otherwise, new nonces are set and the hash is computed again

If there are conflicting blocks (winners), the chain with the most “work” is chosen. Abandoned blocks are called *orphan blocks*.

(→ standard practice of waiting for 6 confirmations to be safe)

The difficulty (or target)

The difficulty is a number (approx. 10^{10} now), which encodes a 32 byte long *target*. The block hash (which can be considered random for all practical purposes) must be below this target for a block to be valid.

Thus the difficulty controls that on average how often there is a winner of the lottery. To have a balanced stream of blocks (in time), this difficulty is adjusted every two weeks (more precisely, every 2016 blocks), based on the number of the blocks in the previous two weeks, so that on average there are 10 blocks (winners) per minute.

While in principle a miner could use a smaller difficulty (higher target) than the others, the protocol says that in case of competing chains (branches), one should choose the one with “more proof-of-work”; more-or-less, this means the one which contains blocks with higher summed difficulty. There is also a standard formula for adjusting the difficulty, but it is done individually by each client.

Note: The difficulty is encoded in the block headers in 4 bytes using a custom exponential encoding.

The blocks on wire (or disk)

If you want to write a blockchain parser, you need to download the blockchain in some format. The official client simply stores the blocks on the disk (in the files `blk00NNN.dat`) more-or-less as they come on the wire, which is:

- ▶ magic bytes `0xf9beb4d9` (4 bytes, big-endian)
- ▶ length of the block (4 bytes, little-endian)
- ▶ the block data as shown before:
 - ▶ header (80 bytes)
 - ▶ number of transactions (varint)
 - ▶ the transactions

Unfortunately, sometimes there is a random number of zero bytes between the blocks. I couldn't find any pattern of where, why, and how many zero bytes are there...

Bitcoin addresses

Bitcoin addresses correspond to private/public keypairs. “Owning” an address is equivalent to knowing the private key. A user can have any number of addresses, and an address can hold any number of “coins”.

Recall that an ECC private key is a random number $d \in [1, n - 1]$, and the corresponding public key is the point $Q = dG \in \mathcal{G}$ on the elliptic curve \mathcal{G} , where $G \in \mathcal{G}$ is the fixed generator. This point can be represented by its affine coordinates $dG = (x, y) \in \mathcal{G} \subset \mathbb{A}^2 \subset \mathbb{P}^2$.

Because of the curve equation $\mathcal{G} = \{ (x, y) : y^2 = x^3 + 7 \}$, the y coordinate can be computed (up to parity) from the x coordinate, giving another representation consisting of x and the *parity of y* .

Unfortunately, because of historical reasons, *both versions* are used on the Bitcoin network, thus there are *two* addresses corresponding to a single private key.

Construction of Bitcoin addresses

- ▶ a representation of the public key is chosen, and encoded as follows
 - ▶ the coordinates are encoded as 32 bytes big-endian integers
 - ▶ full public key: $0x04$ (x coord) (y coord)
 - ▶ compr. pub. key, y even: $0x02$ (x coord)
 - ▶ compr. pub. key, y odd: $0x03$ (x coord)
- ▶ compute $\text{hash} = \text{RIPEMD160}(\text{SHA256}(\text{pubkey}))$
- ▶ add a version byte in the front, which is $0x00$ for the main Bitcoin network ($0x6f$ for testnet3; $0x05$ for pay-to-script-hash addresses)
- ▶ compute $\text{SHA256}(\text{SHA256}(0x00||\text{hash}))$, and discard everything except the first 4 bytes; this will be the checksum.
- ▶ create the string $(0x00||\text{hash}||\text{checksum})$
- ▶ convert this to an ASCII string using Base58 encoding.

Note: This process can be easily inverted to recover the *hash* of the public key, but not the public key itself.

Transactions

A bitcoin transaction consists of several *inputs* and several *outputs*. Inputs refer back to a selected output of an earlier transaction, which this transaction *spends*; and outputs are *conditions* on how to spend them. The creator of a transaction must prove, for each individual input, that he is allowed to spend them.

The most often used condition, called *pay-to-address*, is that the spender must prove that he owns a given address (that is, he knows the corresponding private key).

Another useful transaction is the *generation* or coinbase transaction, which is always the very first transaction in a block, and serves dual purposes:

- ▶ it is a reward for the miner for providing the service of maintaining the network
- ▶ but more importantly, it is a way to “mint” coins. So this is how the supply of bitcoins is created *and* distributed.

There are a few more standard transaction types, and in fact a full scripting language to specify spending conditions, though most of it is disabled (because of security fears).

A randomly chosen transaction (source: <http://blockexplorer.com>)

Hash[?]: 4e1ab2affa72a7393515383a82e60342abb3211c8bbaec8b1c7a6b5572aafb93

Appeared in [block 301742](#) (2014-05-20 13:01:22)

Number of inputs[?]: 3 ([Jump to inputs](#))

Total BTC in[?]: 0.15690866

Number of outputs[?]: 2 ([Jump to outputs](#))

Total BTC out[?]: 0.15680866

Size[?]: 618 bytes

Fee[?]: 0.0001

[Raw transaction[?]](#)

Inputs[?]

Previous output (index) [?]	Amount [?]	From address [?]	Type [?]	ScriptSig [?]
8f97f1623e76....1	0.00311054	1M177PRYquDJ2Ar6anL1qKhhRyabwUi9GV	Address	3046022100d8e3975d9a397284f86faaa2b6d1046d8f773f4702272aa7629b7b8cec4949348af6 ◀ [...] ▶
eea00af26d1b....1	0.04453239	1M177PRYquDJ2Ar6anL1qKhhRyabwUi9GV	Address	3044022047e1bc2c70e34971613db59aff06ad046d8f773f4702272aa7629b7b8cec4949348af6 ◀ [...] ▶
e50edce98c3c....0	0.10926573	1M177PRYquDJ2Ar6anL1qKhhRyabwUi9GV	Address	3045022100cd998cc94904fbf73a01cbbff5e037046d8f773f4702272aa7629b7b8cec4949348af6 ◀ [...] ▶

Outputs[?]

Index [?]	Redeemed at input [?]	Amount [?]	To address [?]	Type [?]	ScriptPubKey [?]
0	Not yet redeemed	0.1065449	1HQUY5fCNHkYBXdkbnhR4ynEX6HJHxTYr	Address	OP_DUP OP_HASH160 b3f290c116ad4d76dfbadb91713438b14d52756 OP_EQUALVERIFY OP_CHECKSIG ◀ [...] ▶
1	Not yet redeemed	0.05026376	1M177PRYquDJ2Ar6anL1qKhhRyabwUi9GV	Address	OP_DUP OP_HASH160 db679dbb781de77866cd2113153544fd78d59f8 OP_EQUALVERIFY OP_CHECKSIG ◀ [...] ▶

A randomly chosen transaction (source: <http://blockexplorer.com>)

```
{
  "hash": "4e1ab2affa72a7393515383a82e60342abb3211c8bbaec8b1c7a6b5572aafb93",
  "ver": 1,
  "vin_sz": 3,
  "vout_sz": 2,
  "lock_time": 0,
  "size": 618,
  "in": [
    {
      "prev_out": {
        "hash": "8f97f1623e768deff46829344fa8358c63074bffa3d1f848cb146ce0f1a44e8f",
        "n": 1
      },
      "scriptSig": "3046022100d8e3975d9a397284f86faaaa2b6d1eae7cde4d4c9690a53b4aefbc94677a956022100e92ac9ba2df27d9e4c9ce9a",
    },
    {
      "prev_out": {
        "hash": "eea00af26d1b6db69dcf29dd3c8b0aae7e4b5627d44b9a48d9ca04de65143d0a",
        "n": 1
      },
      "scriptSig": "3044022047e1bc2c70e34971613db59af6f06ad6913d0b4590e1e83767f2a59f85bb4af002205e72ab9ed4cc9a7aca0b0ba83f:",
    },
    {
      "prev_out": {
        "hash": "e50edce98c3cfa12eb45a16a351cf3924395a1788ab0ec7306eaaa59fa15cf06",
        "n": 0
      },
      "scriptSig": "3045022100cd998cc94904fbf73a01cbbff5e037e796d0954cde6007e702b6e48345139de40220243ca524a8ad6933980fbac6:",
    }
  ],
  "out": [
    {
      "value": "0.10654490",
      "scriptPubKey": "OP_DUP OP_HASH160 b3f290c116ad4d76dfbadb91713438b14d527567 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": "0.05026376",
      "scriptPubKey": "OP_DUP OP_HASH160 db679dbb781de77866cd2113153544fd78d59f8c OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

The structure of a transaction (Tx)

A transaction is basically defined by a list of inputs and outputs. It is encoded as follows:

- ▶ transaction version number (4 bytes; currently `0x00000001`)
- ▶ number of tx inputs (varint¹)
- ▶ list of tx inputs
- ▶ number of tx outputs (varint)
- ▶ list of tx outputs
- ▶ the *lock-time*² (4 bytes)

¹varint = a customly encoded variable-length integer

²The lock-time is a rarely used feature, which specifies a time in the future (measured either in block height or in unix timestamp), before which the transaction cannot be included in any block.

Transaction inputs (TxIn)

A transaction input refers back to a single output of a previous transaction (possibly in the same block), and proves that one is allowed to spend that output. It is encoded as follows:

- ▶ the hash of the previous tx we refer back to (32 bytes)
- ▶ the index of a specific output from that transaction (4 bytes, indexing starts from 0)
- ▶ the length of the *signature script* (varint)
- ▶ the signature script (often called “sigScript”)
- ▶ the sequence number (4 bytes; typically 0)

The signature script contains the proof that we are allowed to spend the given output. In the case of a *pay-to-address* previous tx, it will contain:

- ▶ the *public key* corresponding to the address we own
- ▶ the *signature* with the corresponding *private key* of a modification of *this transaction* (it must be modified to avoid self-signing, which is impossible).

Transaction outputs (TxOut)

A transaction output simply specifies an amount of bitcoins, and a condition of how it can be spent. The most often used condition, called *pay-to-address*, specifies that the spender must own the address given here.

It is encoded as follows:

- ▶ transaction value (8 bytes, encoded as an 8 decimal digits fixed point number)
- ▶ length of the `pubKeyScript`
- ▶ the *pubKeyScript* (or *pkScript*) (containing the condition)

The spending condition is encoded as an arbitrary³ script, which, combined with the future signature script, must produce `true` for the spending tx to be valid.

The sum of the input coins must be equal or more than the sum of the output coins; the difference is the *transaction fee*, which will go to the miner who produces the block which includes the given transaction.

³in practice, the reference client only relays the so-called “standard” scripts

The coinbase (or generation) transaction

The first (and *only* the first) transaction in each block must be the *generation* transaction. This is the only kind of transaction which has no (real) inputs. It serves dual purposes:

- ▶ to reward the miner who produced this block
 - ▶ a constant reward, currently 25BTC, halving every 4 years
 - ▶ plus all the transaction fees of the transactions included in the block
- ▶ to create new bitcoins

The reward is set so that this “minting” of new bitcoins diminishes over the years, and converges to 21,000,000 bitcoins. Thus there will be never more 21M bitcoins (unless this part of the protocol changes, which is really unlikely because none of the holders want that).

There is a field (called *coinbase*) containing arbitrary data; this is used for the extra-nonce, for political messages, the block height, and sometimes to *vote* whether miners accept a proposed protocol change.

Script execution

The script language is a simple⁴, stack-based, non-Turing-complete language.

Execution:

- ▶ first execute the scriptSig (= proof) of *this transaction*. This is not allowed to do anything else than push data to the stack (said data will contain the proof);
- ▶ then delete the secondary stack (which is typically not used anyway);
- ▶ finally execute the pkScript (= condition) of the *previous transaction* (whose output we spend).

If at the end the top of the stack contains **true**, we accept the proof, otherwise not.

Remark: Since each (non-coinbase) tx contains at least one input, which must contain a signature of *this* transaction⁵, third parties cannot modify transactions to redirect the outputs.

⁴there are a *lot* of corner cases, though...

⁵at least with typical spending condition - so called puzzle transaction may not contain such a signature, and thus can be modified by third parties

Pay-to-address script example


step	stack after	opcode	description
0	<i>(empty)</i>	-	initialization
1	sig	PUSH <sig>	signature
2	sig pk	PUSH <pubkey>	pubkey
3	sig pk pk	DUP	duplicate pubkey
4	sig pk addr	HASH160	compute address (= hash of pubkey)
5	sig pk addr pckhash	PUSH <pckhash>	push the recipient address
6	sig pk	EQUALVERIFY	check agreement
7	true	CHECKSIG	check signature

OP_CHECKSIG and signing transactions

The CHECKSIG opcode executes a rather involved procedure.

1. the public key and the signature is popped from the stack, and decoded (we fail if the encoding is invalid).
2. the so-called *subscript* is formed. Usually this is just pkScript⁶
3. extract the so-called *sigHash* byte from the signature (usually this is `SIGHASH_ALL`). Depending on the value of this, we may proceed differently
4. replace all input scripts with empty strings, except the current input which is replaced with the *subscript* (→ tx malleability)
5. append the *sigHash* byte, extended to a 4-byte little endian word, to the serialized, modified transaction
6. verify the signature against the (double-SHA256 hash of) this final bytestring.

The signing procedure is implied by the above checking procedure: The above is done for all inputs do derive all the signatures. For the rest of *sigHash* flag possibilities, the process is a bit different.

⁶but there is a very complicated and confusing procedure of how exactly do this in general, which, as far as I knew, was never executed in history... 

The SigHash byte

The sigHash byte controls how the *outputs* are signed. The above process is slightly modified based on this (for example, the outputs are also modified). This feature can be used for “smart contracts”, which need interaction between different parties to create a transaction. The valid values of the *sigHash* byte are:

- ▶ `SIGHASH_ALL` = `0x01`⁷
- ▶ `SIGHASH_NONE` = `0x02`
- ▶ `SIGHASH_SINGLE` = `0x03`
- ▶ `SIGHASH_ANYONECANPAY` = `0x80` (flag)

`ALL` (the default) means that all *outputs* must be signed.

`NONE` means that none of the outputs should be signed - we don't care where the outputs go.

`SINGLE` means that only the output corresponding to the current input must be signed.

`ANYONECANPAY` means that other people can extend this transaction with more inputs.

⁷but because of some old bug, `0x00` (or a missing byte) is also accepted

Key and signature encodings

Pubkeys are encoded as with the address: 33 or 65 byte, starting with `0x02`, `0x03` or `0x04` (except that `0x06` also appears at some point instead of `0x04`...)

Signatures are $(R, S) \in \mathbb{Z}_n \times \mathbb{Z}_n$ pairs encoded in ASN.1 DER:

$$\text{sig} = [\text{0x30 } N \text{ 0x02 } n_R \underbrace{(\dots R \dots)}_{n_r} \text{ 0x02 } n_S \underbrace{(\dots S \dots)}_{n_s} \text{ sigHash }]$$

$\underbrace{\hspace{15em}}_N$

Except that

- ▶ there is an extra *sigHash* byte at the end
- ▶ DER encoding should be unique, but OpenSSL accepts some non-conforming encodings too, so you have to do that too (→ signature malleability)
- ▶ multisig sometimes looks a bit different

Unfortunately, a client must reproduce all past bugs of the official client (which it has a lot...), as people tested all the bugs by putting transactions in the blockchain. And when they sometimes fix one, you must branch on the block height when they fixed it.

Wallet Import Format: Base58 encoding form of the big-endian 32 byte private key, prepended with a version byte `0x80`; except when the corresponding public key is in compressed form, then also add an extra `0x01` byte at the end.

Signing messages

The official client (was: Bitcoin-Qt, now: Bitcoin-Core) allows signing of custom ASCII messages. This works a bit differently.

First, the message is encoded as $[n_{\text{magic}}||\text{magic}||n_{\text{msg}}||\text{msg}]$, where the lengths are encoded with the *VarInt* encoding, and *magic* is the string "Bitcoin Signed Message:\n".⁸

Then the signature (R, S) , the pubkey format (compressed or uncompressed), the parity of y , and one more bit of extra information (whether $R = x$) is encoded in yet another custom encoding, which is then *Base64 encoded* (not Base58). The extra bits of information are necessary to recover the public key from the signature, so the other party needs to know only your address to check the signature.

The encoding is 1 byte containing the 3 bits (between `0x1b` and `0x22`) followed by 32 byte R and 32 byte S as big-endian integers.⁹

⁸of course this isn't documented anywhere...

⁹...and neither is this

Multisig transactions

There can be transactions which need N out of K signatures to be valid ($N \leq K$), called “multi-sig”. This can be useful in various situations:

- ▶ *two-factor authentication*: with a 2-of-2 multisig transaction, you need both signatures to spend. If the two signing devices are physically separate, this adds extra security (it is not enough to hack one device).
- ▶ *signing rights*: similarly, a company may say that it needs 3 out of 5 executive's digital signature to spend some funds
- ▶ *escrow services*: If there is a seller and buyer on the internet who don't fully trust each other, they can use an escrow. However they both need to trust the escrow. With a 2-of-3 multisig transactions, if there is no dispute, the two parties can sign the transaction; if there is dispute, the escrow can decide and sign it together with the party who is right.

This uses the CHECKMULTISIG opcode, which is even more complex (and contains some new bugs). Usually funds are sent to a *multisig address*¹⁰ first, which can be spent only with the necessary number of signatures.

¹⁰this is pay-to-script-hash (P2SH) address

Pay-to-script-hash (P2SH)

This is an ugly hack to enable reversion of control. It introduces a

- ▶ a new address type (starts with '3'; version byte is `0x05`) (BIP 13)
- ▶ a new “standard transaction type” and a special behaviour for transactions matching that template (BIP 16)

In the address, instead of the 20-byte hash of the public key, we use the 20-byte hash of the *script* which will specify the condition to spend.

The new standard transaction script looks like

```
pkScript = OP_HASH160 [20-byte-hash-value] OP_EQUAL
sigScript = [signatures] {serialized script}
```

If a transaction matches the above pattern, the following special behaviour is invoked:

- ▶ sigScript is executed (fails if there is anything else than OP_PUSH)
- ▶ the serialized script is popped from the stack
- ▶ the hash of this script is checked against the one in pkScript
- ▶ the script is executed

Networking

Bitcoin uses a more-or-less straightforward peer-to-peer networking scheme (based on broadcasting to connected peers). This includes

- ▶ peer discovery
 - ▶ bootstrapping used to be via IRC, these days there are a few wired-in server addresses
 - ▶ then periodically broadcasting our own IP, and asking other nodes for a list of peers
 - ▶ selecting a few peers randomly
- ▶ relaying transactions
- ▶ relaying blocks
- ▶ initial download of older blocks
- ▶ etc

Newer developments:

- ▶ simplified payment verification (SPV) → thin clients
- ▶ some DOS protection is necessary

Elliptic curves

What is an *elliptic curve*?

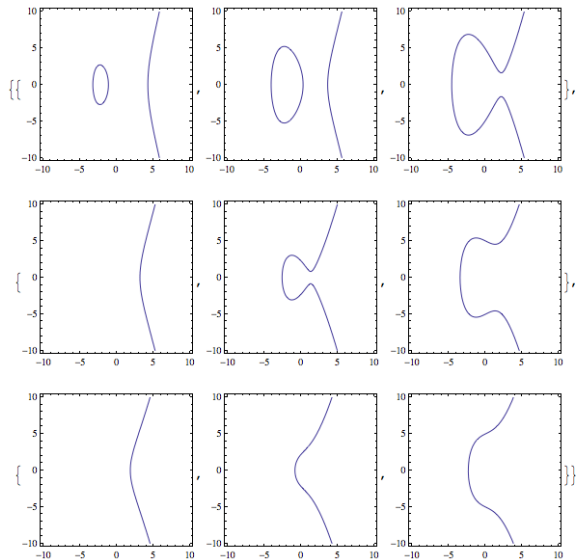
- ▶ $y^2 = x^3 + ax + b$ over a field \mathbb{F}
called the “short Weierstrass form”;
($\text{char}(\mathbb{F}) \neq 2, 3$ and $4a^3 + 27b^2 \neq 0$)
- ▶ smooth cubic plane curve (with a base point)
- ▶ (...other mathematical definitions...)

Etimology / history:

- ▶ arc length of an ellipse
- ▶ elliptic integrals
- ▶ the inverse problem: elliptic functions
- ▶ elliptic curves

Pictures of elliptic curves over \mathbb{R}

Table[ContourPlot[$y^2 = x^3 + a x + b$, {x, -10, 10}, {y, -10, 10}], {a, {-15, -5, +5}}, {b, {-15, +5, +25}}]



The group law on elliptic curves

It's a kind of surprising fact, that elliptic curves has a *group structure*, which furthermore has a rather nice geometric interpretation.

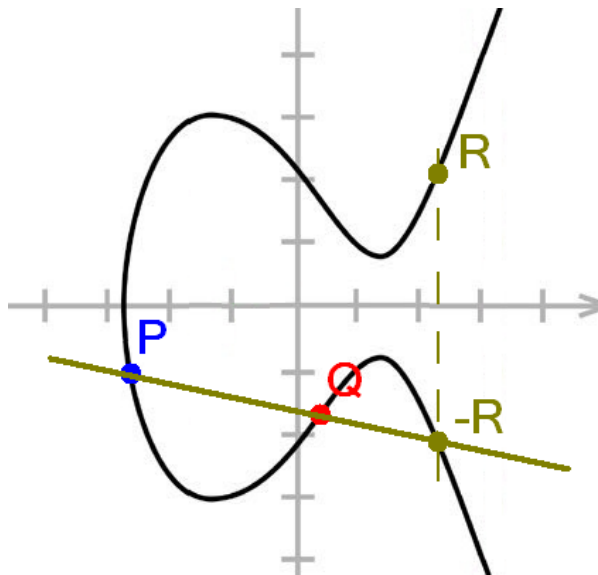
Definitions (for the Weierstrass form):

- ▶ identity element: The point at infinity (denoted by O)
- ▶ inverse: mirroring wrt. the X axis
- ▶ addition: if P , Q and R are on a straight line, we declare $P + Q + R = O$

Group laws:

- ▶ the identity satisfies what it should (trivial)
- ▶ addition is commutative (trivial)
- ▶ addition is associative (nontrivial!)

Addition on elliptic curves



Elliptic curve cryptography

Elliptic curve cryptography is based on the *elliptic curve discrete logarithm problem*. Consider an elliptic curve \mathcal{G} over a (very large) finite field, usually either \mathbb{F}_{2^n} or \mathbb{F}_p for a prime p .

We fix the curve \mathcal{G} and a *generator* $G \in \mathcal{G}$ of this group; that is, $\mathcal{G} = \{kG \in \mathcal{G} : k \in [0, n-1]\}$ where $n = |\mathcal{G}|$ is the size of this group.

- ▶ private key: a random number $d \in [1, n-1] \subset \mathbb{N}$
- ▶ public key: the group element $Q = dG \in \mathcal{G}$

It is easy to compute Q from d and G in such a group, but very hard (at least for appropriate choice of the field and \mathcal{G}) to determine d from Q .

Key length comparison

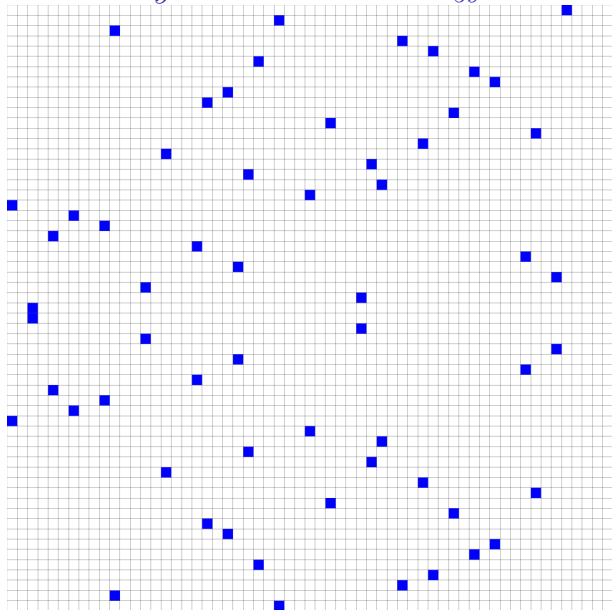
The primary advantage of ECC is that we can have shorter keys for equivalent security:

Security level	RSA key length	ECC key length	ratio
80	1024	160-223	5-6
112	2048	224-255	8-9
128	3072	256-283	11-12
192	7680	384-511	15-20
256	15360	512-571	27-30

The shorter key length can be actually important in practice:

- ▶ embedded devices with limited resources (smartcards, etc)
- ▶ large number of signatures: both bandwidth and storage can be important (bitcoin)

A picture of $y^2 = x^3 + 7$ over \mathbb{F}_{59}



Remark: $p \approx 2^{256} \approx 10^{77} \approx$ no. of elementary particles in the universe

The secp256k1 curve

The safety of ECC depends on the hardness of the elliptic curve discrete logarithm problem. Not all curves are created equal!

Bitcoin uses the curve called secp256k1:

- ▶ the field is \mathbb{F}_p with $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- ▶ the curve equation is $y^2 = x^3 + 7$ (that is, $a = 0$ and $b = 7$)
- ▶ the number of points on the curve n is also a prime, and “close” to p (to be more precise, $p - 2^{129} < n < p - 2^{128}$)
- ▶ since n is prime, the group is cyclic; thus any element (except the infinity) will do as the generator G
- ▶ but there is a concrete, randomly-looking G in the standard

```
p    = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFF2C2F
n    = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
G_x  = 0x 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
G_y  = 0x 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8
```

secp256k1 parameter choices

Arguably, apart from the choice generator, the parameters look sincere:

- ▶ the prime field \mathbb{F}_p is very close to 2^{256} , and differs from it by only a few bits
- ▶ the curve equation: if you look at $y^2 = x^3 + b$ with $b \in \{1, 2, 3, \dots\}$, it seems that $b = 7$ is the very first choice where you can have a somewhat reasonable curve
- ▶ the size of the curve n is also a prime (the co-factor is $h = 1$), which makes the code simpler
- ▶ the generator G , we have no idea how they chose that. On the other hand, the group is cyclic, so in principle any G would do (though it may happen that for special choices of G , the discrete logarithm problem is simpler; I don't know anything about that)

Bernstein's SafeCurves analysis (<http://safecurves.cr.jp.to>)

Curve	Safe?	Parameters:			ECDLP security:				ECC security:			
		field	equation	base	rho	transfer	disc	rigid	ladder	twist	complete	ind
Anomalous	False	True✓	True✓	True✓	True✓	False	False	True✓	False	False	False	False
M-221	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
NIST P-256	False	True✓	True✓	True✓	True✓	True✓	True✓	False	False	True✓	False	False
secp256k1	False	True✓	True✓	True✓	True✓	True✓	False	True✓	False	True✓	False	False
E-382	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
M-511	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
E-521	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓

What does it mean?

- ▶ **disc**: is the discriminant large enough? (small discriminant is not an explicit problem, but the story becomes more complicated, better be safe)
- ▶ **ladder**: is there a simple, fast and *constant-time* scalar multiplication algo (similar to Montgomery ladder)?
- ▶ **complete**: is there a complete addition law (that is, one without special cases like doubling)?
- ▶ **ind**: does an elliptic curve point representation looks random?

Conclusion: secp256k1 is probably OK for what Bitcoin uses it (if you worry about timing attacks, maybe just add random delays?).

Example: Diffie-Hellmann key exchange¹¹

Recall that a *key pair* (d, Q) consists of

- ▶ a private key, which is a random number $d \in [1, n - 1] \subset \mathbb{N}$
- ▶ a public key, which is the group element $Q = dG \in \mathcal{G}$

Let there be two parties: Alice and Bob, with key pairs (d_A, Q_A) and (d_B, Q_B) . They can compute a shared secret $S \in \mathcal{G}$ as follows:

$$d_A Q_B = d_A (d_B G) = \underbrace{(d_A d_B) G}_S = d_B (d_A G) = d_B Q_A$$

Alice can know d_A , so she can compute the leftmost version; Bob knows d_B , so he can compute the rightmost version. But nobody else knows neither d_A or d_B , thus S is their secret.

They can then proceed and use S for any purpose, for example as the key of a symmetric encryption scheme.

¹¹not actually used by Bitcoin

Elliptic Curve Digital Signature Algorithm

Alice writes a message m , and wants to prove that she wrote it. She already has key pair (d_A, Q_A) , and people accept that the public key Q_A in fact belongs to her.

Construction of the signature:

1. compute a hash $z = \text{HASH}(m) \in [1, n - 1]$ of the message m
2. generate an ephemeral key pair: k and $Q_k = kG = (x, y)$
3. let $r = (x \bmod n) \in \mathbb{Z}_n$
4. let $s = k^{-1}(z + rd_A) \in \mathbb{Z}_n$
5. the signature is $(r, s) \in \mathbb{Z}_n \times \mathbb{Z}_n$

Verification of the signature:

1. compute $z = \text{HASH}(m) \in [1, n - 1]$ as before
2. compute $u = s^{-1}z \in \mathbb{Z}_n$ and $v = s^{-1}r \in \mathbb{Z}_n$
3. compute the curve point $(x, y) = Q = uG + vQ_A \in \mathcal{G}$
4. the signature is valid iff $x = r$.

Representations of elliptic curve points

There are many different representations of elliptic curves:

- ▶ Weierstrass affine coordinates \mathbb{A}^2
- ▶ Weierstrass projective coordinates \mathbb{P}^2
- ▶ Weierstrass weighted projective coordinates $\mathbb{P}(2, 3, 1)$
- ▶ Montgomery form
- ▶ Hessian form
- ▶ Jacobian form
- ▶ Edwards form
- ▶ etc...

Representation matters because of efficiency! And possibly also different security properties.

Computations with elliptic curves, I.

How to compute $dQ \in \mathcal{G}$ efficiently, with $d \in \mathbb{Z}_n$ and $Q \in \mathcal{G}$?

Answer: “fast exponentiation”! Write d in binary form:

$d = \sum_{i=0}^{m-1} d_i 2^i$, where $d_i \in \{0, 1\}$.

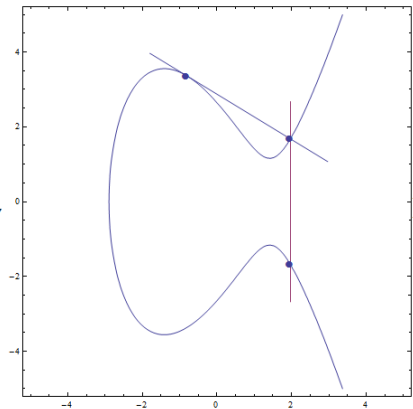
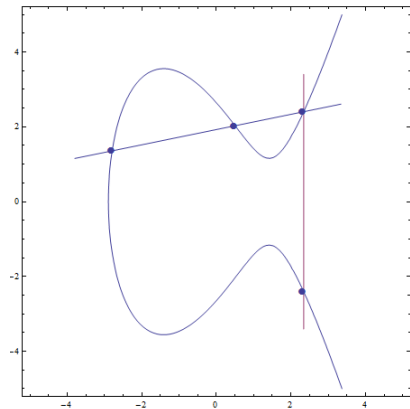
$$dQ = \left(\sum_{i=0}^{m-1} d_i 2^i \right) \cdot Q = \sum_{i=0}^{m-1} d_i (2^i Q) = \sum_{i=0}^{m-1} d_i Q_i$$

where $Q_i = 2^i Q$ can be computed by *repeated doubling*:

$Q_0 = Q$, $Q_1 = 2Q_0$, $Q_2 = 2Q_1$, $Q_3 = 2Q_2$, etc...

Thus we need addition and doubling (which is a special case of addition, but needs to be handled separately anyway).

Elliptic curve addition and doubling in pictures



Elliptic curve addition and doubling in Weierstrass form

In any field \mathbb{F} ($\text{char}(\mathbb{F}) \neq 2, 3$), for two points $P \neq Q \neq O$ on the elliptic curve $y^2 = x^3 + ax + b$, with coordinates $P = (x_p, y_p)$ and $Q = (x_q, y_q)$, it is straightforward (?) to calculate the coordinates of $P + Q = R = (x_r, y_r)$ and $2P = U = (x_u, y_u)$:

$$s = \frac{y_q - y_p}{x_q - x_p}$$

$$t = \frac{3x_p^2 + a}{2y_p}$$

$$x_r = s^2 - (x_p + x_q)$$

$$x_u = t^2 - 2x_p$$

$$y_r = -y_p - s(x_r - x_p)$$

$$y_u = -y_p - s(x_u - x_p)$$

Here s resp. t are the slopes of the secant resp. tangent lines.