Blockchain and Cryptocurrencies

Riccardo Salvalaggio

19th of April, 2021

Contents

1 Introduction **4**

2 Money **4**
- 2.1 Payment Systems . . . 4
- 2.2 Monetary unit . . . 4
- 2.3 Monetary equivalent . . . 4
  - 2.3.1 Fiat Money . . . 4
- 2.4 Monetary Control Structures . . . 4
  - 2.4.1 Creation by Banks . . . 4
- 2.5 Money representations . . . 4
- 2.6 Transaction processing . . . 5

3 Basic Tools **6**
- 3.1 Blockchain . . . 6
- 3.2 Cryptographic Hash Functions . . . 6
  - 3.2.1 Find a collision . . . 6
  - 3.2.2 Hash from Compression . . . 6
  - 3.2.3 Properties and Applications . . . 7

4 Hash Data-Structures **8**
- 4.1 Hash Pointers . . . 8
- 4.2 Hashed Linked List/ Blockchain . . . 8
- 4.3 Merkle Tree . . . 8

5 Cryptography and Digital Signatures **10**
- 5.1 Public key cryptography . . . 10
  - 5.1.1 RSA . . . 10
- 5.2 Digital Signatures . . . 10
- 5.3 Identities . . . 11

6 Two Simple CryptoCurrencies **11**
- 6.1 Goofycoin . . . 11
- 6.2 Scroogecoin . . . 11
  - 6.2.1 Transactions . . . 12
  - 6.2.2 Properties . . . 12

7 Decentralization **12**
- 7.1 Distributed Consensus . . . 12
- 7.2 Incentives and Proof of work . . . 13
- 7.3 Miscellaneous . . . 13

8 Mechanics of Bitcoin **13**
- 8.1 Bitcoin Transactions . . . 14
- 8.2 Bitcoin scripts . . . 15
  - 8.2.1 Pay-to-Public-Key-Hash . . . 15
  - 8.2.2 Pay-to-Publiv-key . . . 15
  - 8.2.3 Multi-Signature . . . 16
  - 8.2.4 Data output/Proof of burn (OP RETURN) . . . 16
  - 8.2.5 Pay-to-Script-Hash . . . 16
- 8.3 Application of Bitcoin scripts . . . 16
  - 8.3.1 Escrow Transactions . . . 16
  - 8.3.2 Micropayments . . . 17
  - 8.3.3 Lock Time . . . 17
  - 8.3.4 Smart Contracts . . . 17
- 8.4 Bitcoin Blocks . . . 17

8.5 Bitcoin Network . . . 17
8.5.1 Network Protocol . . . 18
8.5.2 SPV and Bloom Filters . . . 18

9 Storing and using Bitcoins **19**
9.1 Transmitting Addresses . . . 19
9.2 Storing Bitcoins . . . 20
9.2.1 Wallets . . . 20
9.2.2 Hot and Cold Storage . . . 21
9.3 Splitting and Sharing Keys . . . 21
9.4 Online Wallets and Exchanges . . . 22
9.4.1 Exchanges . . . 22

10 Alternative mining puzzles **22**
10.1 ASIC Resistant Puzzles . . . 22
10.1.1 Scrypt . . . 23
10.2 Proof of useful work . . . 23
10.2.1 Primecoin . . . 23
10.3 Proof of Stake and Virtual Mining . . . 24
10.3.1 Peercoin: Coin Age-Based Selection . . . 24
10.3.2 Tezos: Liquid Proof of Stake . . . 24
10.3.3 Algorand: Pure Proof of Stake . . . 26

11 Bitcoin as a Platform **27**
11.1 Bitcoin as a Log . . . 27
11.2 Smart property . . . 27
11.2.1 Colored Coins . . . 27
11.3 Lotteries . . . 27
11.3.1 Timed Commitment in Bitcoin . . . 28

12 Smart Contracts **29**
12.1 Ethereum . . . 29
12.2 Tezos Smart Contracts . . . 31
12.2.1 Michelson Smart Contracts . . . 32
12.2.2 Michelson Example I: Hello World! . . . 33
12.2.3 Michelson Example II: (Hello paramter!) . . . 34
12.2.4 Michelson Example III: (voting) . . . 34
12.3 Reference Data on Michelson . . . 35
12.4 Unfamiliar Datatypes . . . 36
12.5 TRANSFER_TOKENS . . . 37
12.6 SET_DELEGATE . . . 38
12.7 CREATE_CONTRACT . . . 38

13 Smart Contracts Security **40**
13.1 DAO - Decentralized Autonomous Organization . . . 40
13.2 KotE - King of the Ether . . . 41
13.3 Parity Multisig Bug . . . 42

1 Introduction

Ledger: record of transaction history, centralized, unforgeable.
Blockchain: Decentralized ledger, replicated, need of consensus (blocks because updating at each transaction would be too expensive).
Cryptocurrency: digital money, cryptographic means, rely on blockchain.
Non-monetary Uses: DApps - Decentralized Apps - Smart contracts: storing and executing programs on the blockchain. (e.g.: crypto assets, smart property, voting).

2 Money

Money formalizes the accounting, keeps track of global favor-granting and is useful as memory.

2.1 Payment Systems

A payment system is based **on representation of money, creation of it, and transfer of ownership**. **Cash:** physical, has storage and transport drawbacks.
Electronic: mainly digital, physical token required (e.g.: credit card), book money, centralized point of failure.
Blockchain-based: no physical, algorithmic creation, decentralized consensus.

2.2 Monetary unit

The need of a unit emerges from social processes and ease of interchanging. **Functions:** mean of exchange (w/o money too much exchanging), unit of accounting (allows an interpretation of prices, to compare valuation of goods and in order to have market transparency)and value storage (possibility of save in order to make larger investments).
Properties: durability, transferability, divisibility, homogeneity, verifiability, stability, scarcity.

2.3 Monetary equivalent

Why money has value? Fundamental value (material), payment promise (what you can buy with that), speculation (based on demand for the object and possibility of variation).

2.3.1 Fiat Money

No fundamental value, no payment promise, value entirely based on expectation (no bounds). Most currencies now are Fiat, the value is maintained by a central bank and have legal tender.

2.4 Monetary Control Structures

Two modes of money creation: **Competitive:** (everybody can, limit: production ¡ market value) two possible scenario: gold mining, constant low creation cost, **Monopolized:** (restricted to government agencies) artificially limited.

2.4.1 Creation by Banks

The central bank creates money by loaning to corporate banks. Temporary loans help regulate the money supply. Corporate banks can create money by lending or deposit currency" (to private customers and businesses) that is not backed by legal tender. If all customers withdraws legal tender the bank goes bankrupt.

2.5 Money representations

Physical: physical ownership, easy to use. Bound to location, integrity, divisibility.
Virtual: digital representation of a value not bounded to a real currency. It is nevertheless accepted as a means of payment and can be transmitted, stored and traded. Ownership achieved by cryptographic certificates.

2.6 Transaction processing

Conditions: Capability: transactions can be started and value can be transferred, Legitimacy: only by the owner, Consensus: process to determine the current balance of everyone.

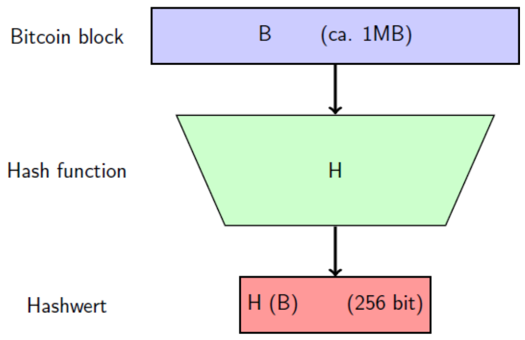
3 Basic Tools

3.1 Blockchain

A blockchain is a growing list of records, called blocks, that are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data.
A transaction is a movement of a monetary item. Transaction history is public, so everyone can know your balance and check every transaction. It's alright unless the history can be rewritten/faked.

3.2 Cryptographic Hash Functions

In the Blockchain, every block consists of the text of the transactions and the hash of the preceding block.



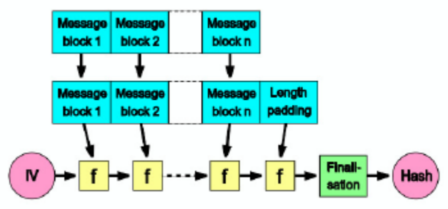
A hash function is a mapping h : B* ->B at n, for some n >0. B = 0, 1.
A compression function is a mapping h : B at m ->B at n, for some m >n >0.
Function h is a one-way function (or preimage resistant) if, given some s in Bn, it is practically impossible to
find a preimage x in D such that h(x) = s.
A collision of h is a pair (x; x0) with x != x0 in D such that h(x) = h(x0).
A function h is second preimage resistant (weakly collision resistant) if, given some x in D, it is practically impossible to
find x0 such that (x; x0) is a collision of h.
A function h is collision resistant if it is practically impossible to fi
nd a collision (x; x0) of h.
Any second preimage resistant function h is also one-way.
Any collision resistant function is also second preimage resistant.(See proofs)

3.2.1 Find a collision

Bitcoin uses SHA-256 so output is B at 256. To find a collision you need more time than universe age. It is practially impossible to
nd a collision (x; x0) for h.
Birthday Paradox: we need to do just N/2 in order ot have 0.5 probability of exploiting the correct answer.

3.2.2 Hash from Compression

Merkle-Damgaard procedure: Let f : Bm ->Bn be a compression function and let r = m - n >= 2. The goal is to construct a hash function h : B* ->Bn from f .
Preprocessing: becomes b'——0 at k——x——0 at r of length t*r.



If f is collision resistant, then the Merkle-Damgaard construction yields a function h that is also collision resistant.

3.2.3 Properties and Applications

Use of MAC for integrity of Data.
- Hiding: consequence of using a one-way. Small inputs can be a problem. To solve: Instead of computing h(x), compute h(rkx) where r is a suitably chosen random number.
- Commitment scheme (MAC scheme): commit and verification. Reqs: Hiding (given com it is infeasible to find msg) Binding (it is infeasible two find two mesages tht return same commit). To enforce, use nonce, number used once: it add perturbation.
- Def. Puzzle friendly: if for every possible n-bit output value and every k chosen from a distribution with high min-entropy, then it is infeasible to and x such that h(k——x) = y in time significantly less than 2 at n.

SHA-256 is defined via the Merkle-Damgaard construction from a compression function the compression function is designed such that flipping a bit in the input changes at least 50% of the bits in the output. Not know to be compromised, but successors exist since 2012: SHA-3 (Keccak).

4 Hash Data-Structures

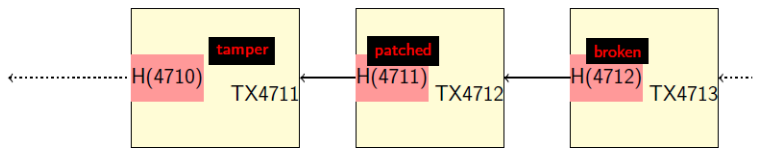
4.1 Hash Pointers

It is a self verifying data structure defined by a couple: (target, h(target)) in order to guarantee data integrity.

```
import hashlib
## library with various hash functions
class HashPointer:
    def __init__ (self, target : bytearray):
        self._target = target
        self._hash = self.digest()

    def verify(self):
        return self._hash == self.digest()

    def digest(self):
        m = hashlib.sha256(self._target)
        return m.hexdigest()

ba = bytearray ("A Few Good Men", 'utf-8')

hp = HashPointer(ba)

hp.verify() ## returns True

ba[6] = 70 ## tamper with the target

hp.verify() ## returns False
```

4.2 Hashed Linked List/ Blockchain

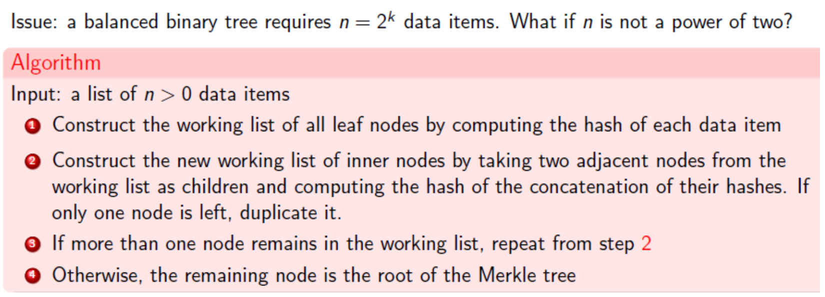
A Blockchain can be seen as a Hashed Linked List in which every transaction in addition to its text, has the hash of the preceding block.
Tampering with data breaks hash in subsequent block Patching that hash breaks hash in next block and so on up to the root which cannot be patched.



```
class Block:
    def __init__ (self, data : bytearray, prev = None):
        self._prev = prev
        self._data = data
        self._prev_hash = prev.digest() if prev is not None else bytearray(256)

class Blockchain:
    def __init__(self):
        self._root_hash = bytearray (256)
        self._list = None
```

4.3 Merkle Tree

A hash tree or Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes.


Issue: a balanced binary tree requires $n = 2^k$ data items. What if n is not a power of two?

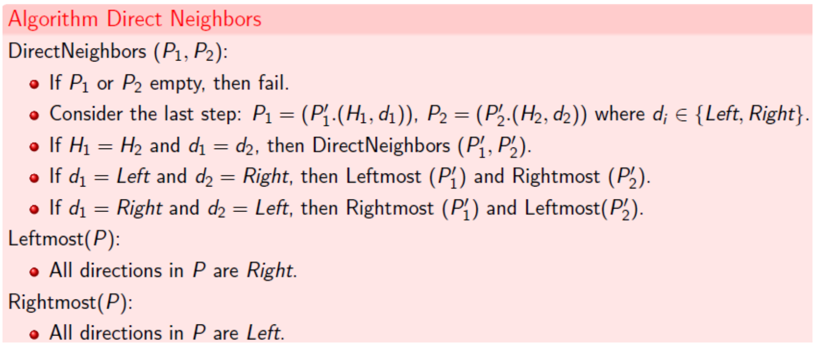
Algorithm

Input: a list of $n > 0$ data items

1. Construct the working list of all leaf nodes by computing the hash of each data item
2. Construct the new working list of inner nodes by taking two adjacent nodes from the working list as children and computing the hash of the concatenation of their hashes. If only one node is left, duplicate it.
3. If more than one node remains in the working list, repeat from step 2
4. Otherwise, the remaining node is the root of the Merkle tree

Figure 1: Construction of a Merkle Tree

The root hash (Top hash) summarizes all information in the tree. It can be useful as proof of membership of a data block in the tree, non-membership in a sorted tree and to ensure temporal order of transactions.
- Proof of Membership: It is sufficient to provide the hash of the data block and all hashes along the path from that data block to the root of the tree. In a tree of size n the proof consists of O(log n) hash values.
- Proof of non-membership: we need:
1. proof of membership of the largest x1 such that x1 ¡ x in the tree
2. proof of membership of the smallest x2 such that x ¡ x2 in the tree
3. demonstrate that x1 and x2 are direct neighbors by comparing the proofs.


Algorithm Direct Neighbors

DirectNeighbors (P_1, P_2):

- If P_1 or P_2 empty, then fail.
- Consider the last step: $P_1 = (P'_1.(H_1, d_1))$, $P_2 = (P'_2.(H_2, d_2))$ where $d_i \in \{Left, Right\}$.
- If $H_1 = H_2$ and $d_1 = d_2$, then DirectNeighbors (P'_1, P'_2).
- If $d_1 = Left$ and $d_2 = Right$, then Leftmost (P'_1) and Rightmost (P'_2).
- If $d_1 = Right$ and $d_2 = Left$, then Rightmost (P'_1) and Leftmost(P'_2).

Leftmost(P):

- All directions in P are $Right$.

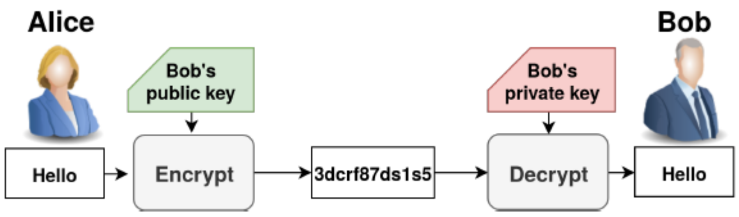
Rightmost(P):

- All directions in P are $Left$.

5 Cryptography and Digital Signatures

5.1 Public key cryptography

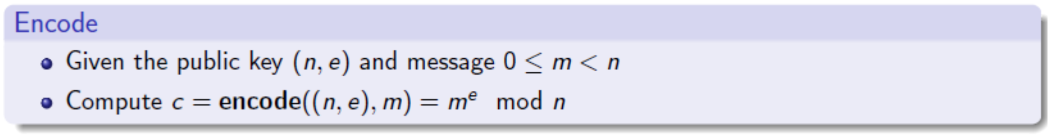
- Symmetric encryption: Encode, Decode. **Decode(k,encode(k,m)) = m.**
Partners needs same key, if it is exposed, confidentiality is compromised.
- Asymmetric encryption: Pair of different keys: Pub_K, Pri_K. Public key is freely shared, the private is known only by the owner. Asymmetric is more expensive than symmetric, so most of the times is only used for key exchanging.
Decode(Pri_K,encode(Pub_k,m)) = m. It is a one-way encryption.



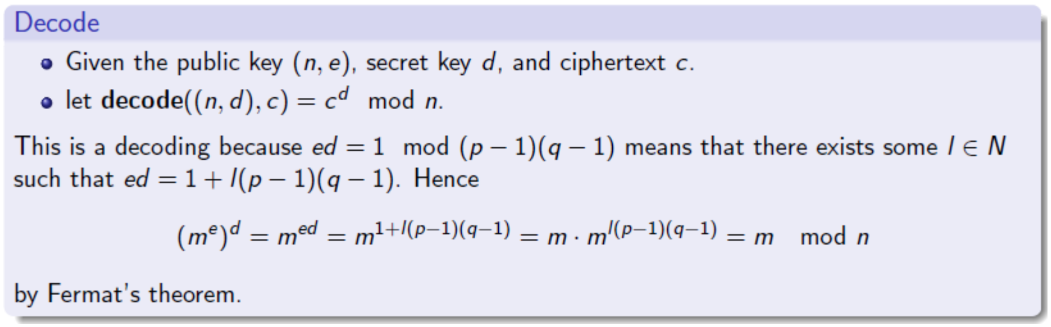
- Cryptosystem: Key pair, keygenerator, encode, decode (polynomial time algorithm).

5.1.1 RSA

Choose two primes p,q: n= pq of k bits.
Choose 1¡e¡(p-1)(q-1): e is coprime with (p-1)(q-1).
Choose 1¡d¡(p-1)(q-1): de=1mod(p-1)(q-1). (thanks to EEA)
Pub_key = (n,e), Pri_key = d.


Encode

- Given the public key (n, e) and message $0 \leq m < n$
- Compute $c = \textbf{encode}((n, e), m) = m^e \mod n$


Decode

- Given the public key (n, e), secret key d, and ciphertext c.
- let $\textbf{decode}((n, d), c) = c^d \mod n$.

This is a decoding because $ed = 1 \mod (p-1)(q-1)$ means that there exists some $l \in N$ such that $ed = 1 + l(p-1)(q-1)$. Hence

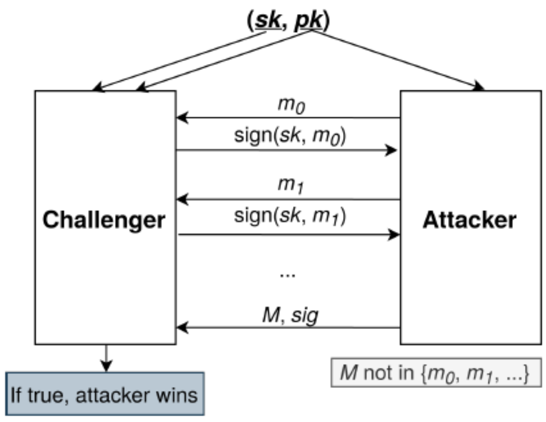
$$(m^e)^d = m^{ed} = m^{1+l(p-1)(q-1)} = m \cdot m^{l(p-1)(q-1)} = m \mod n$$

by Fermat's theorem.

Bitcoin uses a different scheme based on Elliptic curves.

5.2 Digital Signatures

Signature is a handwritten depiction of someone's name, nickname etc.
Functions: proof that signer has seen the content of the document, integrity of the document, signature has to be difficult to forge, but easy to verify.
Same mechanism of Public-key cryptography but sign with private and decrypt with public, so everyone can decrypt but only the owner can sign.



- Practical concerns: many algorithms are probabilistic, limit on message size (possible solution, sign the hash).
Bitcoin uses ECDSA (Elliptic Curve Digital Signature Algorithm) that provide 128 bits: Pri_K = 256 bits, Pub_K = 512 bits (compressed 257 bits), m_size = 256 bits, signature = 512 bits.

5.3 Identities

Identity consists of a pair (Pri,Pub).
Decentralized Identity Management: new identities can be created at any time, good randomness source is required, no need for central user registry, but transactions may reveal behaviour and connections.

6 Two Simple CryptoCurrencies

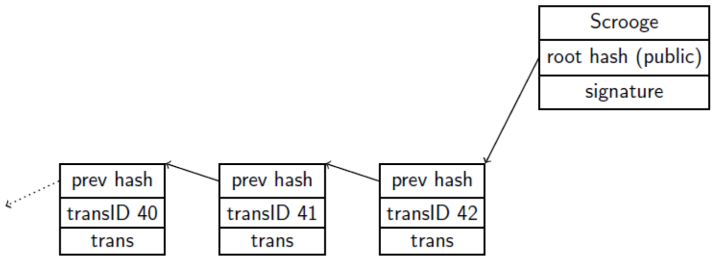
6.1 Goofycoin

Only Goofy can create a coin, whoever own can transfer.
- To create: Generate ID, add to ”CreateCoin”, sign.
- To transfer: create statement ”pay [coin] to [person]”, coin is the hash, person is the public key, sender signs the entire result. This statement now stands for the coin.
- To validate: if ”CreateCoin” verify the signature of Goofy, if ”pay coin to...” the hash is valid, signature verifies previous owner and the hash is received from the previous owner.

This structure has a great drawback since it doesn’t keep track of the transactions (need of Blockchain): ”Double Spending Attack”: Alice signed the transfer twice to two different receipters, the received coin is valid for both.

6.2 Scroogecoin

The basic is as before with the add of an append-only transaction log (Blockchain). Only Scrooge can create coins. Only Scrooge can perform transactions.



6.2.1 Transactions

- CreateCoins:

Need Scrooge's signature.

```
{"transID": 41,
 "type": "CreateCoins",
 "coins_created": [
  {"num": 0, "value": 3.2, "recipient": "C336F45C354D"},
  {"num": 1, "value": 1.4, "recipient": "96B68A43D344"},
  {"num": 2, "value": 7.1, "recipient": "0B7FD2428F5A"}
 ]}
```

- PayCoins:
for each consumed coin, the owner needs to sign off on the transaction.

```
{"transID": 42,
 "type": "PayCoins",
 "coins_consumed": ["41.2", "10.1", "10.0"],
 "coins_created": [
  {"num": 0, "value": 7.1, "recipient": "C336F45C354D"},
  {"num": 1, "value": 2.9, "recipient": "E9DD2D492273"},
],
 "signatures" : [
   "<signature for 41.2>",
   "<signature for 10.1>",
   "<signature for 10.0>"
]}
```

Scrooge appends only valid transactions to the blockchain. Transaction history is valid and safe because everyone can check validity of a transaction and whether it is contained in the blockchain.

6.2.2 Properties

Coins are immutable: cannot be splitted or combined, need to be recreated by another transaction.
In this implementation there is a fundamental problem: There is a single point of failure centralized. Solution: decentralization.

7 Decentralization

The solution to Scroogecoin problem is achieved by Bitcoin organizing the system as a network of equal nodes (i.e., a peer-to-peer network).

7.1 Distributed Consensus

Consensus is needed to synchronize all replicas, data and computation is replicated for reliability, integrity, increase trust etc.
In the consensus we assume that: nodes can communicate and some are faulty / malicious. After finite time, all honest nodes agree on an output value (generated by an honest node). Limits: impossible to achieve consensus if more than 1/3 of the generals are traitors. No algorithm can always reach consensus in bounded time.
- Bitcoin news:
1. Incentives
2. Embracing Randomness
3. No Identities

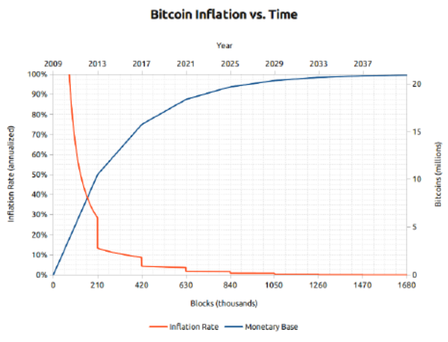
- Implicit consensus:

New transactions are broadcasted to everyone, each round, a random node is selected to broadcast its block. Blocks are accepted if all transactions are valid. If accepted, included.

- Possible Threats:
1. Stealing Bitcoins: Computationally impossible (forge signature).
2. Denial of Service Attack: after some repeated ingores, wait for the next honest.
3. Double spending: only one block will be accepted.

7.2 Incentives and Proof of work

1. Block Reward
Block creators get a reward. Deflation phenomenon: total amount of bitcoins is predetermined, so, it will keep or even increase its value.
2. Transaction Fees
total output of a transaction may be less than the total input and the block creator can cash the difference.
- Proof of work:
Select nodes in proportion to their computing power, set up a competition among all nodes.



- Mining:
Solving hash puzzles competitively to collect the block reward and the transaction fees. Probability that Alice succeeds mining depends on the fraction of global hash power that she controls. mean time to next block = 10 minutes/ fraction of global hash power.

7.3 Miscellaneous

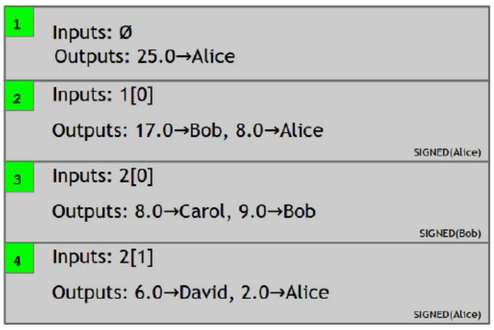
Is mining worth the effort? To answer we should compute the difference between *Mining cost* (hw + operating) and *Mining reward* (block reward + transaction fees).
Orphan blocks: blocks not included in the blockchain (for invalid transactions, attacks, latency).
How do bitcoins obtain value? Factors: security of the blockchain, health of the mining ecosystem, value of the currency, social process of accumulating (relative) trust in the currency.
51 percent attack: even if a single node has the 51% of computational power, he cannot do too much to manage and control maliciously the blockchain.

8 Mechanics of Bitcoin

Bitcoin consensus mechanism gives us an append-only ledger, a data structure that we can only write to. Once data is written to it, it's there forever. There's a decentralized protocol for establishing consensus about the value of that ledger, and there are miners who perform that protocol and validate transactions.

8.1 Bitcoin Transactions

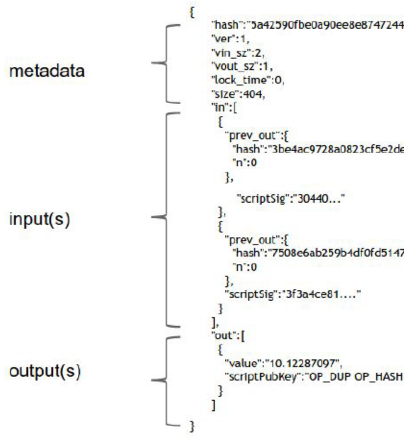
First model: classical account, to move 17 bitcoins to Bob just create a simple transaction with that amount. **Problem:** to determine whether a transaction is valid it is needed to keep track of the account balance. Of course we can make this a little bit more efficient with some data structures that track Alice's balance after each transaction. But that's going to require a lot of extra housekeeping. **Solution:** no account-based model. The ledger just keep track of transactions. Transactions specify a number of inputs and a number of outputs. Each transaction has a unique identifier. If it is a transaction of coin creation, it doesn't require signature.



1	Inputs: Ø Outputs: 25.0→Alice	
2	Inputs: 1[0] Outputs: 17.0→Bob, 8.0→Alice	SIGNED(Alice)
3	Inputs: 2[0] Outputs: 8.0→Carol, 9.0→Bob	SIGNED(Bob)
4	Inputs: 2[1] Outputs: 6.0→David, 2.0→Alice	SIGNED(Alice)

In this example, Alice specifies two outputs, 17 coins to Bob, and 8 coins to Alice. And, of course, this whole thing is signed by Alice, so that we know that Alice actually authorizes this transaction.

- **Change addresses:** Coins are immutable so the entirety of a transaction output must be consumed by another transaction. It could be a different address from the one that owned the 25 bitcoins, but it would have to be owned by her.
- **Efficient verification:** Transactions are easy to validate, we just need to look up the output, make sure of the value and of not double spending. To ensure the last we need to scan until the latest block. Additional data strctures will speed it up.
- **Consolidating funds:** transactions can have many inputs and many outputs, splitting and merging value is easy. In our examples, Bob creates a transaction with the two inputs and one output, with the output address being one that he owns. That lets him consolidate those two transactions.
- **Joint payments:** Say Carol and Bob both want to pay David. They can create a transaction with two inputs and one output, the transaction will need two separate signatures — one by Carol and one by Bob.
- **Transaction syntax:**



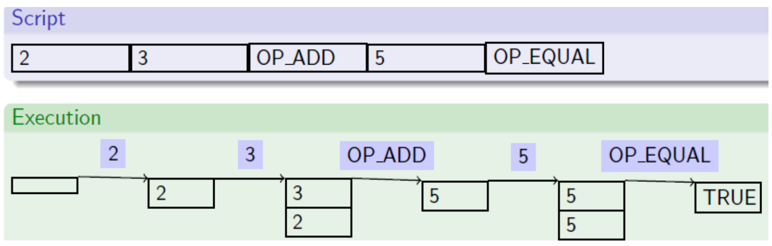
```
{
  "hash":"5a42590fbe0a90ee8e8747244
  "ver":1,
  "vin_sz":2,
  "vout_sz":1,
  "lock_time":0,
  "size":404,
  "in":[
    {
      "prev_out":{
        "hash":"3be4ac9728a0823cf5e2de
        "n":0
      },
        "scriptSig":"30440..."
    },
    {
      "prev_out":{
        "hash":"7508e6ab259b4df0fd5147
        "n":0
      },
      "scriptSig":"3f3a4ce81...."
    }
  ],
  "out":[
    {
      "value":"10.12287097",
      "scriptPubKey":"OP_DUP OP_HASH
    }
  ]
}
```

  - **Metadata** : There's some housekeeping information, There's the hash of the entire transaction which serves as a unique ID for the transaction. That's what allows us to use hash pointers to reference transactions.

- **Inputs:** An input specifies a previous transaction, so it contains a hash of that transaction. The input also contains the index of the previous transaction's outputs that's being claimed.
- **Outputs:** Each output has just two fields. They each have a value, and the sum of all the output values has to be less than or equal to the sum of all the input values.

8.2 Bitcoin scripts

the inputs also contain scripts instead of signatures. To validate that a transaction redeems a previous transaction output correctly, we combine the new transaction's input script and the earlier transaction's output script. We simply concatenate them, and the resulting script must run successfully in order for the transaction to be valid. These two scripts are called scriptPubKey and scriptSig. The key design goals for Script were to have something simple and compact, yet with native support for cryptographic operations. The scripting language is stack-based. This means that every instruction is executed exactly once, in a linear manner.

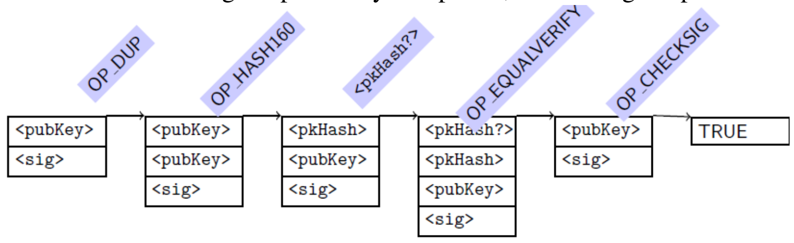


Name	in	out	Function
OP_DUP	1	2	duplicate top of stack
OP_HASH160	1	1	apply SHA-256 to top of stack a RIPEMD-160 hash function
OP_EQUALVERIFY	2	0	continues if the top two items on sta otherwise stops marks transaction a
OP_CHECKSIG	2	1	expects the public key and the (scr ture on the stack; returns TRUE if t is valid and FALSE otherwise
OP_CHECKMULTISIG	$2 + m + n$ $n \leq m$	1	expects the number m followed by the number n followed by n signat stack; returns TRUE if every signa validated with one of the keys

Miners do not accept arbitrary scripts, but they have a whitelist, best practice has narrowed down the scripts in use.

8.2.1 Pay-to-Public-Key-Hash

Assume the unlocking script already completed, the locking script starts on the stack left behind by the unlocking script.



8.2.2 Pay-to-Publiv-key

Most simple form of payment, Needs more space as P2PKH because the hash is shorter, still used in coinbase transactions by miners.

P2PK scripts

`<sig>`	unlocking script from input `scriptSig`

`<pubKey>`
`OP_CHECKSIG`
	locking script from UTXO's `scriptPubkey`

8.2.3 Multi-Signature

Used to redeem a payment where m out of n signers are sufficient.

Locking script

```
M <PubKey 1> <PubKey 2> ... <PubKey N> N OP_CHECKMULTISIG
```

Unlocking script

```
OP_0 <Sig 1> <Sig 2> ... <Sig M>
```

- `OP_0` is due to an implementation error that cannot be amended

Example N=3, M=2

- Locking: `2 <PubKey 1> <PubKey 2> <PubKey 3> 3 OP_CHECKMULTISIG`
- Unlocking: `OP_0 <Sig A> <Sig B>`

8.2.4 Data output/Proof of burn (OP RETURN)

Create transaction output with 0 BTC and locking script OP RETURN, comes with 40 bytes of data storage, usually a SHA-256 hash (32 bytes) plus some tag, trying to spend this output => running OP_RETURN => script ends immediately => transaction invalid.

8.2.5 Pay-to-Script-Hash

A use case of Multi-signature. Multi-Signature is very powerful but awkward to use: customers (that send payments) need special locking scripts/ specialized software, the transaction is about five times larger than a simple payment large UTXO puts burden on full nodes. **P2SH** is the solution: replaces the complex locking script by its hash. Safer: needs to present the script matching the hash (the redeem script) in addition to the unlocking script.

Direct use of Multi-Signature

locking script: `2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG`

unlocking script: `Sig1 Sig2`

Using P2SH

locking script: `OP_HASH160 <20-byte hash of redeem script> OP_EQUALVERIFY`

unlocking script: `Sig1 Sig2 <redeem script>`

redeem script: `2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG`

To validate must run in sequence: unlocking, locking, redeem script

Properties:

- Shorter fingerprints
- Scripts can be encoded as addresses
- Recipient constructs the script
- Data no longer stored in UTXO
- Recipient carries transaction fees

presently any valid script is allowed, except P2SH and OP RETURN.

8.3 Application of Bitcoin scripts

8.3.1 Escrow Transactions

Scenario: Alice orders goods from Bob, but she only wants to pay once the goods are received in good order.
Solution: Alice and Bob set up a 2-of-3 Multi-Signature scheme with Conor, a mediator. Alice makes her payment using Multi-Signature (more likely P2SH), if Alice and Bob agree, they can redeem the payment together. In case of dispute, the mediator has to decide.

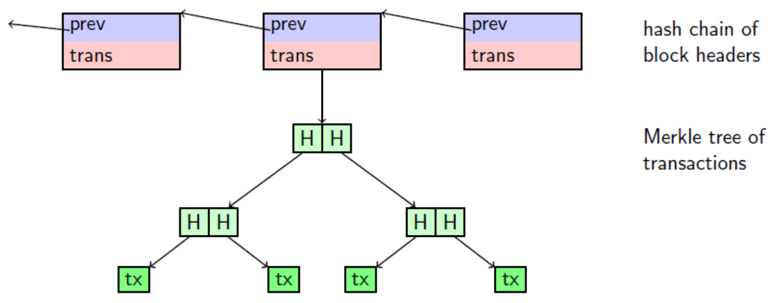
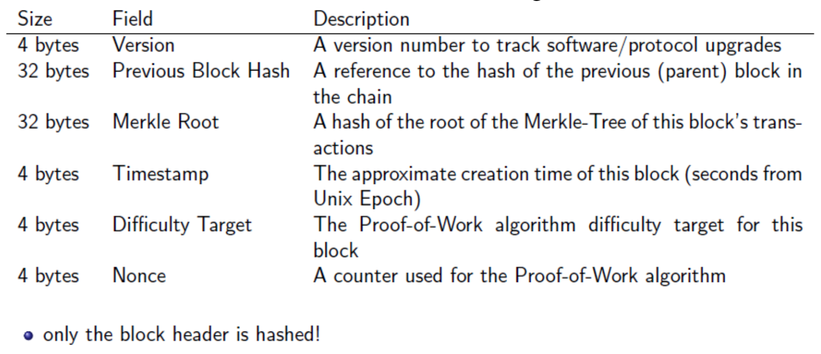


Figure 2: Block Structure



Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the Merkle-Tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The Proof-of-Work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

- only the block header is hashed!

Figure 3: Block Header

8.3.2 Micropayments

Scenario: having many small transaction is not efficient.
Solution: Clark sends a Multi-Signature (for Lois and Clark) transaction paying a deposit. Whenever Clark uses the service, he signs a transaction that pays the acrued charges to Lois and returns the rest to himself. He sends these transaction to Lois. If Clark wants to quit, Lois co-signs the last transaction, submits it, and stops the service.

8.3.3 Lock Time

Scenario: In the previous Micropayments scenario, if Lois never co-signs any transaction, then the deposit is lost for Clark.
Solution: Lois and Clark sign a transaction that returns the deposit to Clark. This transaction contains a lock time, which indicates the earliest time that the transaction can be submitted. This time (block number) serves as a deadline for Lois to cash the charges. If Lois charges in time, then the return transaction would be rejected as double spending.

8.3.4 Smart Contracts

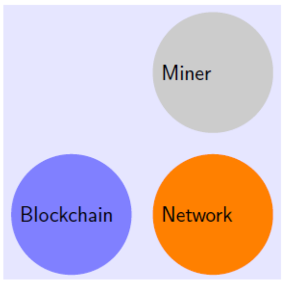
Even though Bitcoin scripts are quite limited, they already enable some interesting applications. Other blockchains have taken this idea further, most prominently Ethereum. Their scripting languages are more powerful (sometimes Turing complete). Scripts are often called smart contracts.

8.4 Bitcoin Blocks

Transaction are grouped in block of 1 MB mainly for efficiency reasons: it keeps the chain shorter, and maintain a reasonable transaction rate (that take 10 minutes).

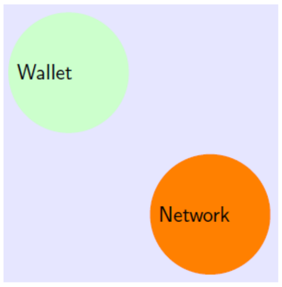
8.5 Bitcoin Network

Peer-to-peer network, it consists of peers (nodes) having equal rights => no notion of servers or clients.



- Alternative: miner farms
- clients do just mining and networking
- coordinated by a (full node) server
- miner farms use a different protocol

Figure 4: Solo miner



- SPV = Simplified Payment Verification
- only block headers are kept locally, not the full chain
- ⇒ 1000x decrease in storage
- do not have full picture of all UTXOs
- need to obtain transaction data from surrounding full nodes to verify, but is prone to double spending attacks
- asking for specific transactions can compromise anonymity ⇒ Bloom filters

Figure 5: Light Weight (SPV) Wallet

- Node types and roles

- **Wallet:** safe keeping of private keys for end users
- **Miner:** minting new bitcoins by creating new blocks.
- **Blockchain:** verification of block traffic, keeping a copy of the blockchain.
- **Network:** routing messages, maintaining connectivity needed for every node.

8.5.1 Network Protocol

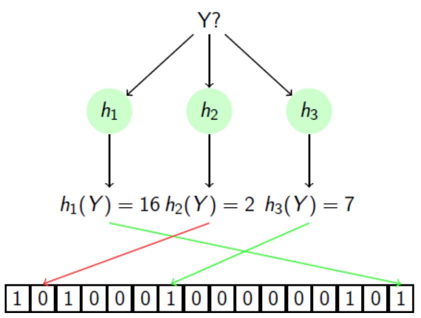
8.5.2 SPV and Bloom Filters

SPV nodes need to retrieve blocks / transactions in connection with the payment they are processing. This selective download can reveal the addresses in the node's wallet. To avoid revelation, SPV nodes request transaction data using Bloom filters. A Bloom filter narrows down the set of transactions without revealing the selection criterion. **- Bloom filters:** A Bloom filter bf is a probabilistic data-structure that provides a fuzzy encoding of a set. Its operations are: **add(x)**, **elem(x)**.
- Implementation of a Bloom filter:

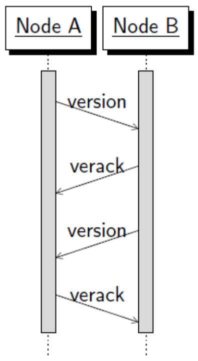
- bitvector of size m, initialized to all 0
- k different independent hash functions $h_i : A \to \{0, \dots, m-1\}$
- to add x
  - compute $y_i = h_i(x)$ for all k hash functions
  - set bits $y_1, \dots, y_k$
- to check (potential) presence of x
  - compute $y_i = h_i(x)$ for all k hash functions
  - return False if any of the bits y_i is 0
  - otherwise return True
- the false positive rate

$$(1 - e^{-kn/m})^k$$

depends on the number n of elements that have been added to the filter



- Bloom filter with three hash functions
- add A
- add B
- check for X
- may be an element
- check for Y
- definitely not an element

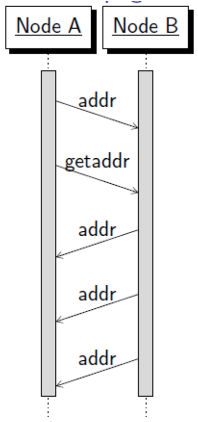


the version message for initial contact with at least one bitcoin node

```
{"PROTOCOL_VERSION": 70015,
 "nLocalServices": 1033,
 "nTime": 1591903131,
 "addrYou": "122.51.104.28",
 "addrMe": "82.38.163.188",
 "subver": "/Satoshi:0.20.0/",
 "BestHeight": 634259
}
```

second part is optional

Figure 6: Network discovery



- When B receives A's getaddr message, it randomly selects up to 2500 recent addresses from its pool and sends them to A
- A connection times out after 90 minutes

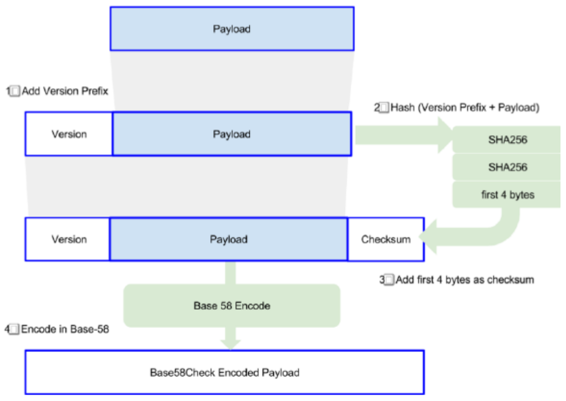
Figure 7: Address propagation

Every node maintains a transaction pool of unconfirmed, validated transactions. They represent the consensus of the longest chain.

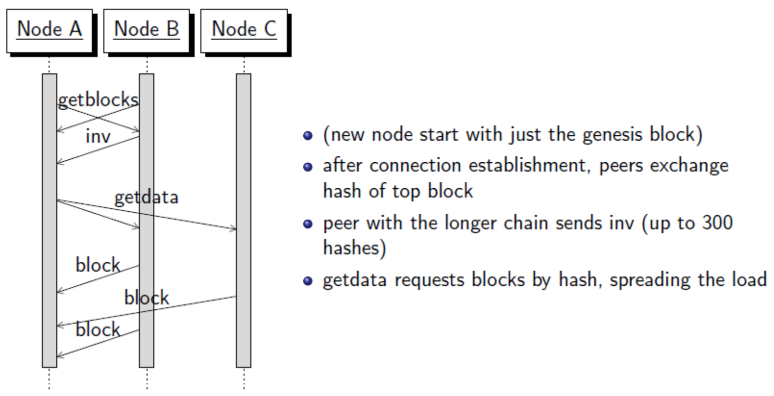
9 Storing and using Bitcoins

9.1 Transmitting Addresses

Bitcoin addresses correspond to public keys, mostly encoded in **BASE-58**: it is a textual encoding of binary data, more efficient than hexadecimal encoding. The type of encoding is denoted by the version prefix in the address.

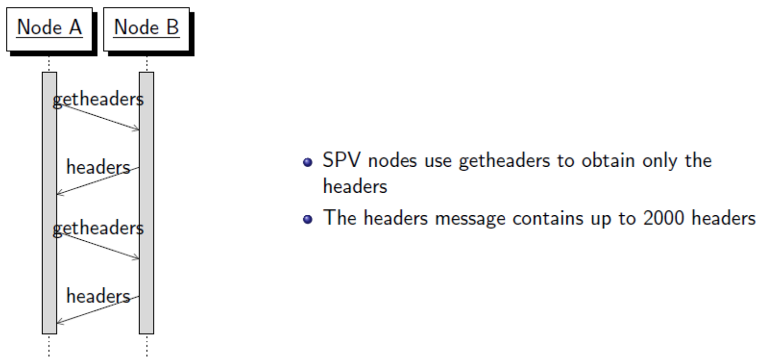


Since a meaningless address is not so practicable, Bitcoin uses QR Codes.



- (new node start with just the genesis block)
- after connection establishment, peers exchange hash of top block
- peer with the longer chain sends inv (up to 300 hashes)
- getdata requests blocks by hash, spreading the load

Figure 8: Network synchronization



- SPV nodes use getheaders to obtain only the headers
- The headers message contains up to 2000 headers

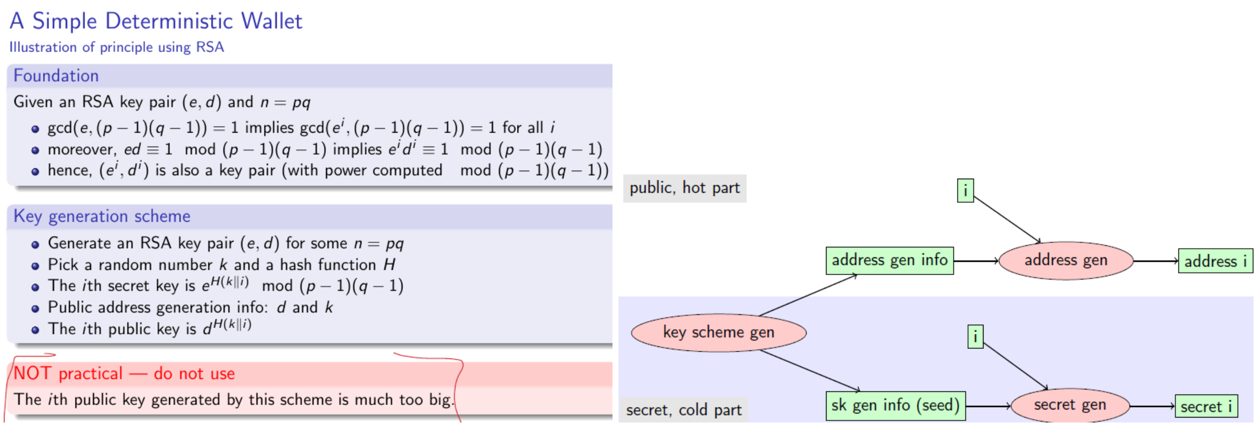
Figure 9: Header Synchronization

9.2 Storing Bitcoins

To transact with Bitcoins we need **public info:** addresses (to send and receive), **secret info:** secret key to sign (to send), scripts matching the hases of P2SH (to receive. There are three main aspects in storing Bitcoins: Availability, Security, Convenience (hard to achieve all of them).

9.2.1 Wallets

- **Wallet software:** management of key, UTXOs, change addresses, receiver addresses.
- **Random Wallets:** Non-deterministic wallets, have pre-generated random private keys. **Drawbacks**: need of remembering keys, backups.
- **Deterministic Wallets**: starting from a single random seed, addresses are generated algorithmically, only the seed needs to be remembered (and kept secret).



A Simple Deterministic Wallet

Illustration of principle using RSA

Foundation

Given an RSA key pair (e, d) and $n = pq$

- $\gcd(e, (p-1)(q-1)) = 1$ implies $\gcd(e^i, (p-1)(q-1)) = 1$ for all i
- moreover, $ed \equiv 1 \mod (p-1)(q-1)$ implies $e^i d^i \equiv 1 \mod (p-1)(q-1)$
- hence, (e^i, d^i) is also a key pair (with power computed $\mod (p-1)(q-1)$)

Key generation scheme

- Generate an RSA key pair (e, d) for some $n = pq$
- Pick a random number k and a hash function H
- The ith secret key is $e^{H(k\|i)} \mod (p-1)(q-1)$
- Public address generation info: d and k
- The ith public key is $d^{H(k\|i)}$

NOT practical — do not use

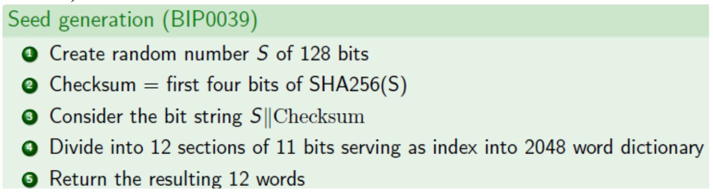
The ith public key generated by this scheme is much too big.

A simple example can be done with RSA (not so practical, keys become too big).**Real scenario:** tree structure to express additional organizational meaning, e.g. - one branch for incoming payments, - another branch for change from outgoing payments. For key generation, **Child Key Derivation** is used: *The chain code is used to introduce seemingly random data to the process, so that the index is not sufficient to derive other child keys. Thus, having a child key does not make it possible to find its siblings, unless you also have the chain code. The initial chain code seed (at the root of the tree) is made from random data, while subsequent chain codes are derived from each parent chain code .*

9.2.2 Hot and Cold Storage

- **Hot storage:** private keys stored in the wallet sw, full availability and convenience, but security concerns
- **Cold storage:** private keys stored in a safe place offline. to access UTXOs we need to import the key previously.

How to manage cold storage in order to maintain anonymity and addresses management? Use deterministic seeded (stored offline) wallet.



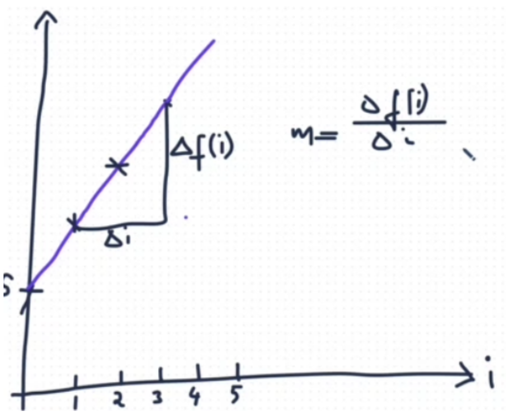
Seed generation (BIP0039)

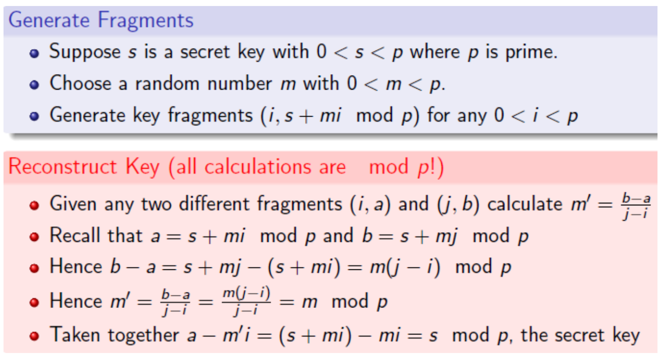
1. Create random number S of 128 bits
2. Checksum = first four bits of SHA256(S)
3. Consider the bit string $S\|\mathrm{Checksum}$
4. Divide into 12 sections of 11 bits serving as index into 2048 word dictionary
5. Return the resulting 12 words

- **Paper wallets**: Bitcoin private keys printed on paper, secured against hackers, if generated offline and never stored on a computer.

9.3 Splitting and Sharing Keys

Another approach to secure storing of a secret key, divide the key into several parts and store them separately. But from any n ¿ 1 parts, it is possible to reconstruct the key.





Generate Fragments

- Suppose s is a secret key with $0 < s < p$ where p is prime.
- Choose a random number m with $0 < m < p$.
- Generate key fragments $(i, s + mi \mod p)$ for any $0 < i < p$

Reconstruct Key (all calculations are $\mod p$!)

- Given any two different fragments (i, a) and (j, b) calculate $m' = \frac{b-a}{j-i}$
- Recall that $a = s + mi \mod p$ and $b = s + mj \mod p$
- Hence $b - a = s + mj - (s + mi) = m(j - i) \mod p$
- Hence $m' = \frac{b-a}{j-i} = \frac{m(j-i)}{j-i} = m \mod p$
- Taken together $a - m'i = (s + mi) - mi = s \mod p$, the secret key

It works because Z_p is a field.

Last problem to avoid reconstruction is in signing, where the key must be present. **Solution:** Threshold Cryptography: perform partial signature with a key fragment, signature completed once a sufficient number of partial signatures have been applied. (Also can be done with multi-signatures, P2PKH is better).

9.4 Online Wallets and Exchanges

Another way to store Bitcoins is in Wallets in the Browser, simple installation and multiple devices availability, but keys are store with the service provider.

9.4.1 Exchanges

An exchange is like a bank, for each customer it keeps the account in several different cryptocurrencies and offers a trading service. The trade is not a transaction. Only when withdrawing from the exchange is it necessary to have a wallet. **Convenience:** connection FIAT - Crypto. **Risks:** Bank run: too many customers demands their money at the same time. Ponzi schemes may run the exchange, payouts are performed using deposit of new customers. Hacker attack: if the exchange is honest, it controls bitcoin addresses with large amounts.

10 Alternative mining puzzles

Mining puzzles are at the core of Bitcoin: determine the incentive system, limit the ability to control consensus.
- Puzzle requirements

- **reasonable difficulty**
- **fast verification**
- **asjustable difficulty**
- **progress fee:** probability of winning a puzzle should be roughly proportional to hash power.

Bitcoin's example: puzzle solutions are found at a fairly predictable rate (roughly 10 minutes).

10.1 ASIC Resistant Puzzles

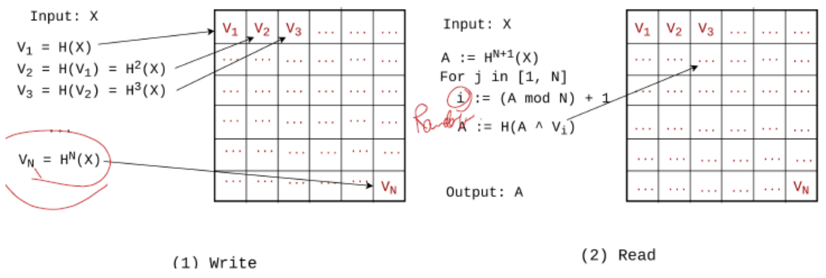
More efficient, it results in a major performance gain over normal hw mining. **- Goal:** allow ordinary computers to mine, prevent someone to dominate.
Main idea (memory-hard): design mining puzzles that require a large amount of memory to solve. the performance of memory is more stable than processors through the years.

10.1.1 Scrypt

Most famous memory-hard mining puzzle used in many cryptocurrencies: same puzzle condition, use memory-hard function instead of SHA-256, requires fixed amount of memory.

```
def scrypt(H, N, X):
    // initialize memory buffer of length N
    V = [0] * N

    // Fill up memory buffer with pseudorandom data
    V[0] = H(X)
    for i in range(1, N):
        V[i] = H(V[i−1])

    // Access memory buffer in a pseudorandom order
    A = H(V[N−1])
    for i in range(N):
  // choose a random index based on A
        j = A % N
  // update A based on this index
        A = Hy
        (A ^ V[j])
    return A
```

Fill a large buffer of random access memory with random values. Read from this memory in a pseudorandom order.



- Disadvantages: Scrypt trades memory for computation speed, verification requires the same amount of memory/-computation: N steps and N memory to check the correctness of the proof.

10.2 Proof of useful work

Consensus protocol not designed for doing useful computations, is just a waste of energy. Look for a puzzle that is useful for the solution of practical problems.

10.2.1 Primecoin

Puzzle is based on finding chains of prime numbers.
- Cunningham chain:
A Cunningham chain of length k is a sequence p1, p2, p3,...,pk where each pi is a large prime number and $p_i = 2p_{i-1} + 1$. Given a challenge x (the hash of the previous block), take the first m bits of x valid solution: any chain of length ≥ k where the first element is an n-bit integer and has the same m leading bits as x.

10.3 Proof of Stake and Virtual Mining

- Disadvantage of Proof of Work

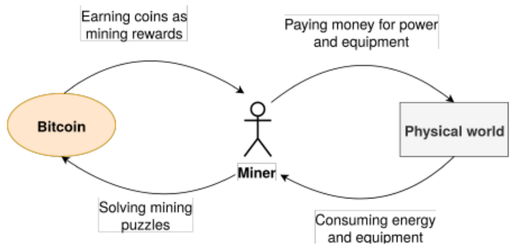
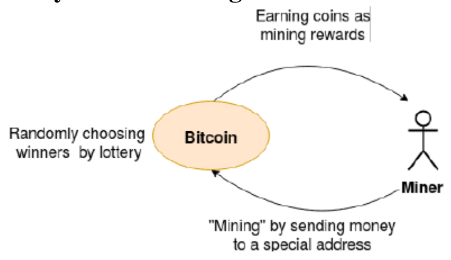

Earning coins as mining rewards
Paying money for power and equipment
Bitcoin
Miner
Physical world
Solving mining puzzles
Consuming energy and equipment

Figure: Loop on mining in Proof of work algorithms

- Why Virtual mining


Earning coins as mining rewards
Randomly choosing winners by lottery
Bitcoin
Miner
"Mining" by sending money to a special address

remove the wasteful half of the proof of work mining cycle.
- Proof of Stake
Each coin holder has stake in the coin system, the next miner is random, size of the stake determine the probability to be chosen.

10.3.1 Peercoin: Coin Age-Based Selection

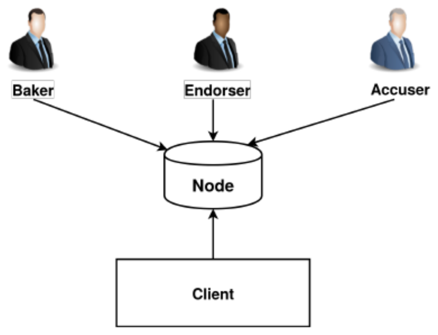
age of a coin: time since last use of the coin.
Miners must solve a SHA-256-based computational puzzle, but the difficulty of this puzzle is lowered for older coins.

10.3.2 Tezos: Liquid Proof of Stake

Tezos is a 3rd generation blockchain featuring:

- blockchain and cryptocurrency
- expressive contract language
- live-upgrade of protocol and governance

(PoS) assigns minting power based on the proportion of coins held. Besides minting there is validation also based on stake. It is liquid because coin holders can delegate their coins to others for minting or validating.
- Trust model: Bakers deposit coins as stake (security deposit), if he accept invalid transactions he loses coins.



- Tezos Architecture:

- **Node:** The local component of the system, it manages the context and sync with the network, communicate with other peers, connect: endorser, baker, accuser and client.
- **Client:** Interface to the node
- **Baker:** Bakes (creates) new blocks.
- **Endorser:** Verifying the validity of a block and agreeing on a block by endorsing that block.
- **Accuser:** Monitors all blocks and looking for invalid transactions.

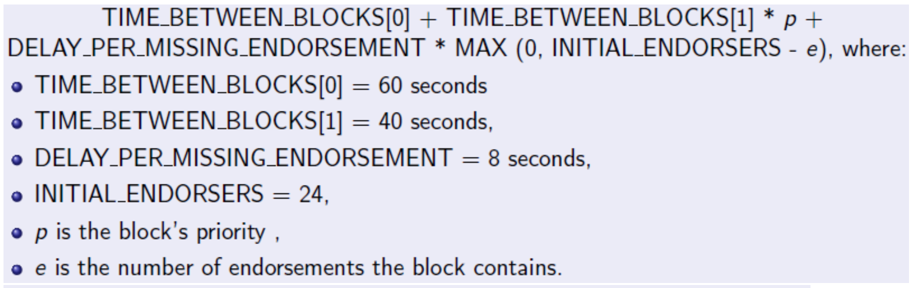
Everything's organized in cycles, Incentivize: rewarded for baking and endorsing, a security deposit is frozen and could be released or burnt if double-baking or double-endorsment.
- Cycles:
Blocks are group into cycles of $BLOCK\_PER\_CYCLE$= 4096 blocks.
$TIME\_BETWEEN\_BLOCKS$ = one minute: (1 cycle = 2 days, 20 hours, 16 minutes).
$PRESERVED\_CYCLES$ = 5 cycles (14 days, 5 hours, 20 minutes).

- Rolls: A roll represents a set of coins delegated to a given key: $TOKENS\_PER\_ROLL$ = 10000 tokens (8000 currently).
Each delegate has a stack of roll ids. Roll snapshots are taken every $BLOCK\_PER\_ROLL\_SNAPSHOT$ = 256 blocks (16 times per cycle). Rolls are useful to determine baking and endorsement rights.

- Delegations:
a baker becomes passive for cycle n: if it failed to create any blocks or endorsements in the past $PRESERVED\_CYCLES$ cycles A passive delegate cannot be selected for *baking* or *endorsing*.
A baker has to hold at least one roll (owned or delegated), always the same probability to earn baking rights in the system.

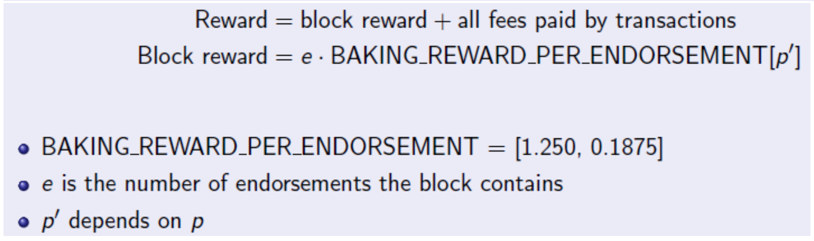
- Baking in Tezos: The net randomly selects a priority list as follows.
Priority0 = Roll 6
Priority1 = Roll 9
Priority2 = Roll 4
Priority3 = Roll 5
...
Priority9 = Roll 7

If *Roll 6* does not create and broadcast a block within 1 minute, the owner of Roll 9 may take over, and so on. A block is valid only if its timestamp has a minimal delay with respect to the previous block's timestamp.



$$\text{TIME\_BETWEEN\_BLOCKS}[0] + \text{TIME\_BETWEEN\_BLOCKS}[1] * p + \text{DELAY\_PER\_MISSING\_ENDORSEMENT} * \text{MAX}\ (0, \text{INITIAL\_ENDORSERS} - e), \text{ where:}$$

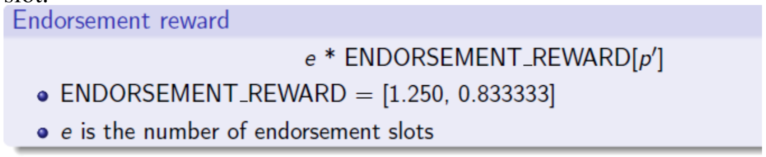
- TIME_BETWEEN_BLOCKS[0] = 60 seconds
- TIME_BETWEEN_BLOCKS[1] = 40 seconds,
- DELAY_PER_MISSING_ENDORSEMENT = 8 seconds,
- INITIAL_ENDORSERS = 24,
- p is the block's priority ,
- e is the number of endorsements the block contains.



$$\text{Reward} = \text{block reward} + \text{all fees paid by transactions}$$
$$\text{Block reward} = e \cdot \text{BAKING\_REWARD\_PER\_ENDORSEMENT}[p']$$

- BAKING_REWARD_PER_ENDORSEMENT = [1.250, 0.1875]
- e is the number of endorsements the block contains
- p' depends on p

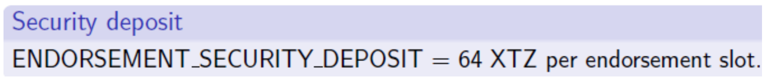
$BLOCK\_SECURITY\_DEPOSIT$ = 512 XTZ per block created.
frozen for $PRESERVED\_CYCLES$ = 5 cycles.
$ENDORSERS\_PER\_BLOCK$= 32 endorsers by randomly selecting active rolls.
It verifies the last block baked (at level n) and emits an endorsement operation baked in block n + 1. Once block n + 1 is baked => no other endorsement for block n will be considered valid. An endorser may have more than one endorsement slot.



Endorsement reward

$$e * \text{ENDORSEMENT\_REWARD}[p']$$

- ENDORSEMENT_REWARD = [1.250, 0.833333]
- e is the number of endorsement slots



Security deposit

ENDORSEMENT_SECURITY_DEPOSIT = 64 XTZ per endorsement slot.

- Delegation: Nodes can delegate coins to a baker
if they have not enough XTZ, or do not want to set up computing infrastructure to bake blocks. The baker cannot spend delegated coins, it has a higher probability of being selected, the baker shares the additional revenue with the coin holder.
- Fork Choice Rule: The canonical chain based on the number of bakers that endorsed the block, at every block height, 32 random rolls are selected to endorse a block. The block with the most endorsements is treated as the canonical one.

10.3.3 Algorand: Pure Proof of Stake

It has the convenience and efficiency of a centralized system without the poblems of current decentralized implementations. Based on **ALGOrithmic RANDomness** (immune from manipulations) to select veri
ers in charge of constructing the next block. No different types of users. Consensus based on fast algorithm for Byzantine agreement: probability of forks very small (10^{-18}).

- Algorand Proof of Stake Algorithm

- **1. Proposal**
  A token is taken randomly chosen among all tokens. The owner of this token proposes, signs, and broadcast a new block.

- **2. Agreement**
  1000 tokens are randomly chosen among all tokens, the owners agree.

- Self-selecting: Each user runs his/her own lottery, cannot cheat, but can prove the winning. Winners broadcast their winning tickets and their agreements about the proposed block.
- Techniques: the players of BA* are selected to be a much smaller subset of the set of all users, each new block B^r will be constructed and agreed upon, via a new execution of BA* by a separate set of selected verifiers SV^r. The last block B_{r-1} is used to determine the next verifier set as well as the leader in charge of constructing the new block B_r. Q_r is unpredictable, users have a special role in the generation of the rth block.
Q_{r-1} contained in B_{r-1} => adversary might immediately corrupt all verifier and the leader. the leader for the next block, secretly assembles the proposed new block and then disseminates it for certification with proof-of-leadership.

- Properties: amount of computation is minimal, new block generated in less than 10 minutes, will never leave the blockchain (no fork, almost certainly), all power resides with the users themselves.

11 Bitcoin as a Platform

How to use Blockchain and Bitcoin for non monetary goals.

11.1 Bitcoin as a Log

Blockchain is an append-only tamperproof ledger that keep track of its history.
One of the most important property that it must have is a secure Timestamping that prove the succession of event:

Choose random r with high min-entropy, publish $H(r||x)$ at time T, use OP_RETURN or *coin burn*, if challenged, we can produce r and x and point to the record on the blockchain.
As of 2014, the preferred way to do Bitcoin timestamping is with an OP_RETURN transaction which results in a provably unspendable output. The OP_RETURN instruction returns immediately with an error so that this script can never be run successfully, and the data you include is ignored. **- Applications:** prior knowledge of ideas, proof of submission, DS schemes.

11.2 Smart property

As we know you can trace ownership of value in the Bitcoin system over time, simply by following the transaction graph. This is bad for anonymity, since you can often track ownership of coins this way. This leads to the fact that **bitcoins aren't fungible.** In economics, a fungible good is one where all individual units are equivalent and can be substituted for one another. For example, **gold is fungible** since one ounce of (pure) gold can be substituted for any other ounce of gold. Instead, due to the past history, a 1.0 bitcoin is not the same as another 1.0 bitcoin. . For example, just as coin collectors value old coins, some day bitcoin collectors might place special value on coins originating in the genesis block or some other early block in Bitcoin's history..

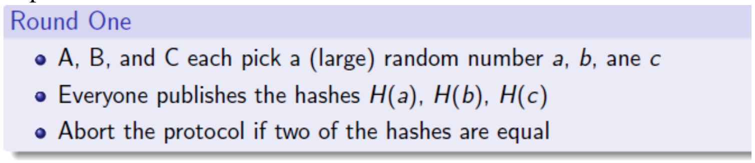
11.2.1 Colored Coins

Examples of colored coins (Bitcoins with metadata): tickets, shares, collectibles, subscriptions.
The main idea is to stamp some Bitcoins with a "color", and track that color stamp even as the coin changes hands, just like we are able to stamp metadata onto a physical currency. To achieve this, in one transaction, called the "issuing" transaction, we'll insert some extra metadata that declares some of the outputs to have a specific color.
We could have another bitcoin transaction that takes several inputs:
some green coins, some purple coins, some uncolored coins, and shuffles them around. It can have some outputs that maintain the color. There may need to be some metadata included in the transaction to determine which color goes to which transaction output. We can split a transaction output of four green coins into two smaller green coins.

11.3 Lotteries

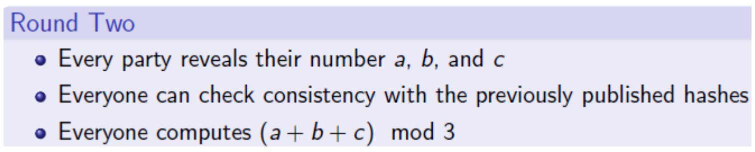
Problems of Bitcoin Lotteries are that there is no random instruction in the scripts, betters are not physically present, simultaneity cannot be guaranteed on internet, etc.

A possible solution can be this scenario:



Round One

- A, B, and C each pick a (large) random number a, b, ane c
- Everyone publishes the hashes $H(a)$, $H(b)$, $H(c)$
- Abort the protocol if two of the hashes are equal



Round Two

- Every party reveals their number a, b, and c
- Everyone can check consistency with the previously published hashes
- Everyone computes $(a + b + c) \mod 3$

Drawback: a player could choose to never reveal his number and block the protocol indefinitely. **Solution:** automatic loss.

11.3.1 Timed Commitment in Bitcoin

Alice puts up a bond that vouches for value x, a bitcoin transaction with an output that can be spent in two ways: Signed by Alice and Bob or that includes value x but only Alice's signature.
Then, A and B sign and the bond is transferred to Bob, the trans has a lock time t, A intends to reveal x before t so it will never be accepted, if A succeds her bond is returned, otherwise her bonds falls to Bob.
Algorithm

Locking script for the output

```
OP_IF
  <AlicePubKey> OP_CHECKSIGVERIFY <BobPubKey> OP_CHECKSIG
OP_ELSE
  <AlicePubKey> OP_CHECKSIGVERIFY OP_HASH <H(x)> OP_EQUAL
OP_ENDIF
```

Unlocking script; bond forfeited

```
<BobSignature> <AliceSignature> 1
```

Unlocking script; bond returned

```
<x> <AliceSignature> 0
```

- Special case: *n* party lottery
Timed hash commitments, $n^2 - n$ commitments required, players have to escrow more than they are betting. Alice puts up a bond, in the form of a Bitcoin transaction output script that specifies that it can be spent in one of two ways. One way is with a signed transaction from both Alice and Bob. The other way to spend it is with a signature from just Alice, but only if she also reveals her random number. If Alice's random string is x, then the scriptPubkey actually contains the value H(x).
Now if Alice leaves without revealing her value, Bob can claim the bond at time t. This doesn't force Alice to reveal her commitment but she will lose the entire bond that she put up.

Locking script for the output

```
OP_IF
  <AlicePubKey> OP_CHECKSIGVERIFY <BobPubKey> OP_CHECKSIG
OP_ELSE
  <AlicePubKey> OP_CHECKSIGVERIFY OP_HASH <H(x)> OP_EQUAL
OP_ENDIF
```

Unlocking script; bond forfeited

```
<BobSignature> <AliceSignature> 1
```

Unlocking script; bond returned

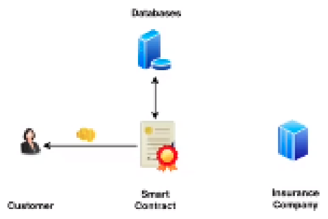
```
<x> <AliceSignature> 0
```

12 Smart Contracts

Traditional contracts: contract parties, common goal, obligation of each party, clauses, will.
A **smart contract** is a program stored and executed on a blockchain, It is code (Code is Law) and can self-executing and self-enforcing rules.
- Goals: Trusted execution without third parties, security and low costs.
- Properties: Immutable because on the blockchain, **distributed** because it's validated all over the network, **transparent:** consensus about the result.
- Functions: send and receive coins, interact with other contracts.
- Examples:

- *Crowdfunding:*
  If succeed transfer the money, otherwise return to the investors.

- *Flight Delay Insurance:*
  Ensure that a customer is compensated if the flight delays for more than two hours, the smart contract is linked to the databses that record the flight status.



- Storing:
Everything is stored on the blockchain (limited by cost).
- Gas:
Normally, fixed fee to reward the miner (on Bitcoin it depends on the size of the transactions). For **smart contracts** it depends on the *effort*: time to run, storage. Each operation require **Gas**: Total fee = $gas\_Used * gas\_Price$. Anyway, there is a limit (deposited like a prepayment) when the gas runs out: network stops to execute, contract goes back to its original state. If the limit is too low, fee still to be paid even if the execution fails, If it is too high: unused gas is returned.

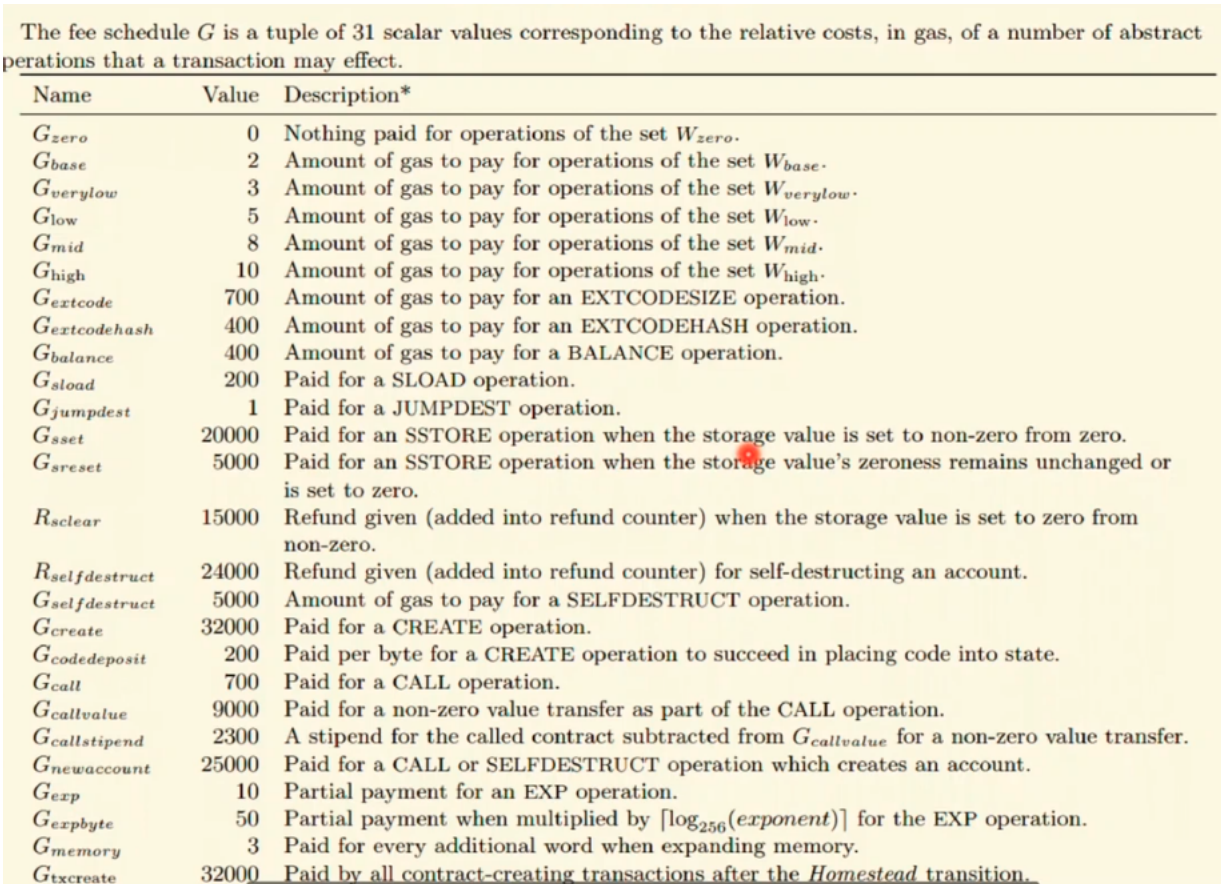
12.1 Ethereum

First and most well-known smart contracts blockchain, use a Blockchain 2.0 (Distributed computing platform). **Goal:** decentralized sw dev platform. (Initially 1 Ether = $10^{1}8$).
- Features:
Block time about 15 seconds, **minting** constant rate, **mining:** pow but different algorithm Ethash for which ASICs are less advantageous, but is planning to switch to PoS, **transactio fees** are calculated by effort, bandwidth use and storage, finally Ethereum is **account-based**(managed with a private key or a smart contract), coins are represented by account balances.
Pros and cons of account-based

- **Advantages:** More flexible transactions, dependance on state and external input allows for oracles and other logic to influence the outcome. Transactions are smaller.

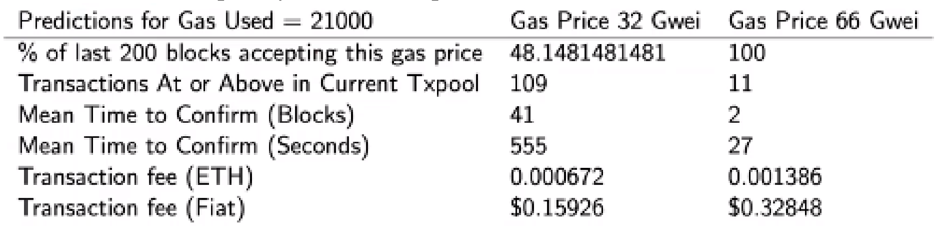
- **Disadvantages:** Transactions for the same account are scheduled, the account models encourage address reuse that compromise privacy.

Ethereum Fee Schedule


The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract perations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for an EXTCODESIZE operation.
$G_{extcodehash}$	400	Amount of gas to pay for an EXTCODEHASH operation.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead* transition.

Gas Price
Gas price determines how quickly the network processes the contract. The miner decides whether accept the transaction or not.



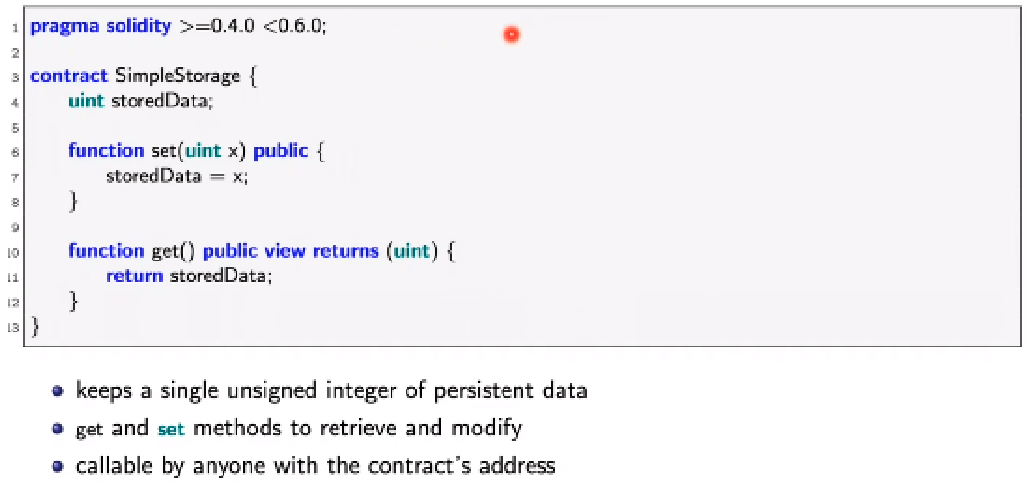
Predictions for Gas Used = 21000	Gas Price 32 Gwei	Gas Price 66 Gwei
% of last 200 blocks accepting this gas price	48.1481481481	100
Transactions At or Above in Current Txpool	109	11
Mean Time to Confirm (Blocks)	41	2
Mean Time to Confirm (Seconds)	555	27
Transaction fee (ETH)	0.000672	0.001386
Transaction fee (Fiat)	$0.15926	$0.32848

Ethereum Virtual Machine
Execution platform for all smart contracts, Stack-Based but Turing-complete, has special instructions for Hashing, Signing etc. **Components:** account state, world state, storage state, block info, runtime environment.

- **Accounts**
  two types: Externally owned accounts, controlled by private key, and contract accounts associated with contract code.
  components of account state: Nonce, Balance (in Wei), CodeHash.
- **World state**: mapping address $->$ account state. Maintained in Modified Merkle Patricia Trie.
- **Storage state**: account specific information at run time.
- **Block information**:
- **Runtime environment information**:

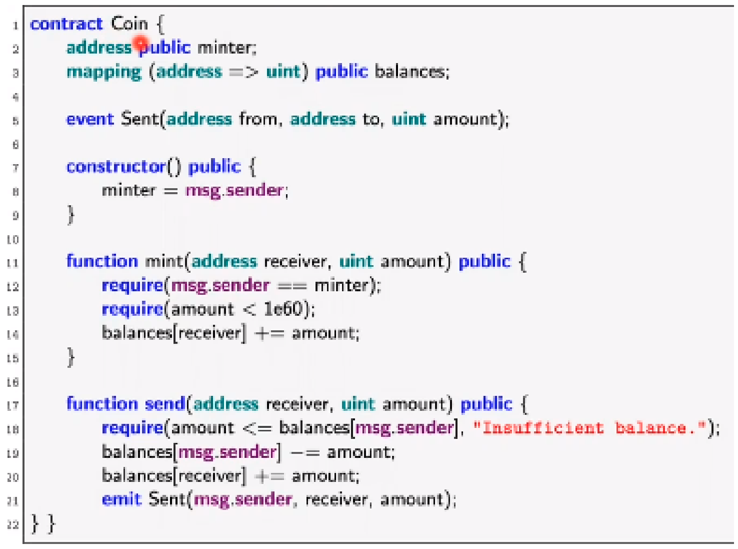
- High-Level Contract Languages
Solidity (most-common), Serpent, LLL, Vyper.

- Solidity
OO, high-level, influenced by C++, python and JS, Statically typed, inheritance, libraries etc.
Code example:



```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

- keeps a single unsigned integer of persistent data
- get and set methods to retrieve and modify
- callable by anyone with the contract's address



```
contract Coin {
    address public minter;
    mapping (address => uint) public balances;

    event Sent(address from, address to, uint amount);

    constructor() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
} }
```

- Challenges: privacy and security: no effective way to guarantee the security of smart contracts. **performance:** gas and store limits, limited resources. **programming languages and tools limitations**.
- Smart contracts benefits: transparency, automated, accuracy, security, speed, efficiency, trust, clear communication, storage & backup, paper free.

12.2 Tezos Smart Contracts

- Tezos: decentralized blockchain platform that supports smart contracts. Blockchain 3.0: live-upgrade, on-chain governance. PoS, mathematicallt verified components.

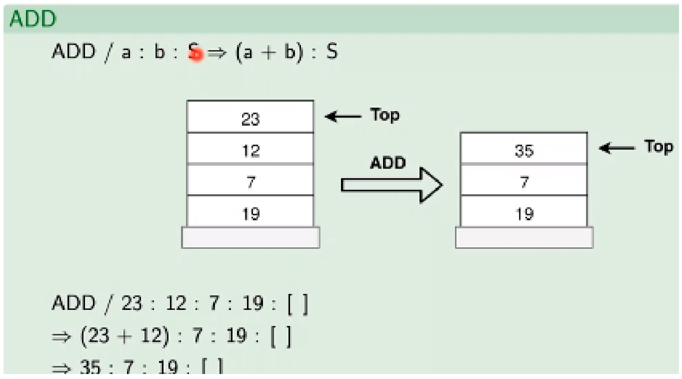
- **- On-chain governance:** a goverance is any system for managing and implementing changes to cryptocurrency blockchains. In On-chain governance, rules for instituting changes are encoded into blockchain protocol. Developers propose changes through code updates and each node votes on whether to accept or reject the proposed change.
- **- Background:** changes are proposed by developers who seek consensus with the main stakeholders (developers and miners).

- Examples for On-Chain Governance

- Tezos: self-amending ledger, changes are rolled out on a test chain.
- Dfinity: mission: builnding the world's biggest virtual computer based on blockchain, goal: more flexible tha "code is law". Hardcoded constitution, triggers passive and acive actions.

So, in Tezos' platform, contracts are registered together with a private data storage, they are executed by performig transactions to their associated account, data passed as parameters and viewed as procedure call. It's account based and uses **Michelson** as language.

- Michelson:
stack-based, high-level data types, strict static type checking. Contract execution cannot fail because an instruction has been executed on a stack of unexpected length or contents. It is also based on sequences of instructions, input: stack resulting from the previous instruction, rewrites it to the next one, all values are immutable and garbage collected. **There are no local variables, but just values on the stack**.

- Example:

ADD

ADD / a : b : S ⇒ (a + b) : S

ADD / 23 : 12 : 7 : 19 : [ ]
⇒ (23 + 12) : 7 : 19 : [ ]
⇒ 35 : 7 : 19 : [ ]

12.2.1 Michelson Smart Contracts

The type describes the values on the stack before and after the instruction.
- example: DUP :: 'a: 'S − > 'a: 'a: 'S. 'a stands for any type, 'S stands for any Stack.
- example: DUP; ADD :: 'a: 'S − > 'a: 'a: 'S ; int: int: 'S − > int: 'S. It works only if 'a = int.
DUP; ADD :: int: 'S − > int: int: 'S ; int: int: 'S → int: 'S
the intermediate types fit together
DUP; ADD :: int: 'S − > int: int: 'S ; int: int: 'S → int: 'S
...can be elided
DUP; ADD :: int: 'S − > int: 'S
DUP; ADD :: a : S − > (a+a) : S

- Contract types:
contract :: (Parameter p, Storage s) − > ([Operation], Storage s)
contract is implemented by a sequence I of Michelson instructions
I :: (pair 'Parameter 'Storage) : [ ] − > (pair (list operation) 'Storage) : [ ]

12.2.2 Michelson Example I: Hello World!

```
parameter unit;
storage string;
code { DROP;
        PUSH string "Hello World!";
        NIL operation; PAIR;
     };
```

type of input stack :: (pair unit string) : [ ]

Running with storage '""' and input Unit

initial stack value :: (Pair Unit "") : [ ]

initial stack type :: (pair unit string) : [ ]

```
code { DROP;
        PUSH string "Hello World!";
        NIL operation; PAIR;
     };
```

- **DROP instruction** drops the top element of the stack: stack :: DROP / $_{:}s- > S$
- **PUSH instruction** adds a value with a certain type onto the top of the stack: stack :: PUSH 'a x / S − > x : S.
- **NIL instruction** adds an empty list of a certain type onto the top of the stack. stack :: NIL 'a / S − > [] : S
- **PAIR instruciton** removes the top two elements of the stack and pushes the pair onto the stack. stack :: PAIR / a:b:S − > (Pair a-b):S

END

```
storage
  "Hello World!"
emitted operations [ ]
```

Type check

```
Well typed
{ parameter unit ;
  storage string ;
  code { /* [ pair (unit @parameter) (string @storage) ] */
         DROP
         /* [] */ ;
         PUSH string "Hello World!"
         /* [ string ] */ ;
         NIL operation
         /* [ list operation, string ] */ ;
         PAIR
         /* [ pair (list operation) string ] */ } }
```

12.2.3 Michelson Example II: (Hello paramter!)

Another example

```
parameter string;
storage string;
code { CAR;
          PUSH string "Hello ";
          CONCAT;
          NIL operation; PAIR;
       };
```

Type checking

```
{ parameter string ;
  storage string ;
  code { /* [ pair (string @parameter) (string @storage) ] */
         CAR
         /* [ @parameter string ] */ ;
         PUSH string "Hello "
         /* [ string, @parameter string ] */ ;
         CONCAT
         /* [ string ] */ ;
         NIL operation
         /* [ list operation, string ] */ ;
         PAIR
         /* [ pair (list operation) string ] */ } }
```

CAR instruction

select left component of pair

stack value :: CAR / (Pair a _) : S $\mapsto$ a : S

stack type :: pair 'a 'b : 'S $\to$ 'a : 'S

CONCAT

string concatenation

stack :: CONCAT / a : b : S $\mapsto$ $a \uparrow b$: S

type :: string : string : 'S $\to$ string: 'S

where $a \uparrow b$ concatenates strings a and b (alternative writing a^b)

Running on storage '""' and input '"Tezos"'

```
storage
  "Hello Tezos"
emitted operations
```

12.2.4 Michelson Example III: (voting)

An open vote with a fee, fixed list of choices, identities not stored.
- **Types:** storage: map string int. parameter: string.

Verify that the caller sent enough tokens to be able to vote. If not, make the call fail

```
code{
#(name, storage)
AMOUNT;
# amount : (name, storage)
PUSH mutez 5000;
# 5000 : amount : (name, storage)
IFCMPGT{PUSH string "stingy!"; FAILWITH}
{};
# (name,storage)
```

AMOUNT pushes the number of tokens received, IFCMPGT is a macro: compares the two numbers on top (removing them in the process) if the top number was greater, executes its first branch (code in braces).

```
DUP;
# (name, storage) : (name, storage)
UNPAIR;
# name : storage : (name, storage)
GET;
# (Some current | None) : (name, storage)
```

UNPAIR destructs the pair to get a stack with the key on top and the map beneath, GET consumes the name and storage map on top to lookup the name in the map.

```
IF_SOME
    {
      # current : (name, storage)
      PUSH int 1; ADD;
      # current+1 : (name, storage)
      SOME}
    {PUSH string "Unknown super computer";
     FAILWITH};
    # (Some (current + 1)) : (name,storage)
```

if the count is some integer value, add 1 to this value, if not, fail because of an unknown name.

Reorder the elements and then update the map

```
# (Some (current + 1)) : (name, storage)
DIP{UNPAIR};
# (Some (current + 1)) : name : storage
SWAP;
# name:(Some (current+1)) : storage
UPDATE;
# updated storage
```

DIP applies the code in braces one element below the stack top, SWAP exchanges the two top elements of the stack, UPDATE updates the map with the new count

Return updated storage

```
NIL operation;
PAIR;
# (nil, updated storage)
```

Type check

```
Well typed
{ parameter string ;
  storage (map string int) ;
  code { /* [ pair (string @parameter) (map @storage string int) ] */
         AMOUNT
          /* [ @amount mutez : pair (string @parameter) (map @storage string int) ] */ ;
          PUSH mutez 5000
          /* [ mutez : @amount mutez : pair (string @parameter) (map @storage string int) ] */ ;
         IFCMPGT
            { PUSH string ”stingy!”
              /* [ string : pair (string @parameter) (map @storage string int) ] */ ;
             FAILWITH
              /* [] */ }
            { /* [ pair (string @parameter) (map @storage string int) ] */ }
          /* [ pair (string @parameter) (map @storage string int) ] */ ;
DUP
/* [ pair (string @parameter) (map @storage string int)
   : pair (string @parameter) (map @storage string int) ] */ ;
UNPAIR
/* [ @parameter string : @storage map string int
   : pair (string @parameter) (map @storage string int) ] */ ;
GET
/* [ option int : pair (string @parameter) (map @storage string int) ] */ ;
IF_SOME
  { /* [ @some int : pair (string @parameter) (map @storage string int) ] */
    PUSH int 1
    /* [ int : @some int : pair (string @parameter) (map @storage string int) ] */ ;
    ADD
    /* [ int : pair (string @parameter) (map @storage string int) ] */ ;
    SOME
    /* [ option int : pair (string @parameter) (map @storage string int) ] */ }
  { PUSH string ”Unknown super computer”
    /* [ string : pair (string @parameter) (map @storage string int) ] */ ;
FAILWITH
     /* [] */ }
 /* [ option int : pair (string @parameter) (map @storage string int) ] */ ;
 DIP { /* [ pair (string @parameter) (map @storage string int) ] */
       UNPAIR
       /* [ @parameter string : @storage map string int ] */ }
 /* [ option int : @parameter string : @storage map string int ] */ ;
 SWAP
 /* [ @parameter string : option int : @storage map string int ] */ ;
 UPDATE
 /* [ @storage map string int ] */ ;
 NIL operation
 /* [ list operation : @storage map string int ] */ ;
 PAIR
 /* [ pair (list operation) (map @storage string int) ] */ } }
```

12.3 Reference Data on Michelson

- Core data types

Data Type	Description
string	strings
nat	natural numbers
int	integers
bytes	bytes
bool	booleans (True or False)
unit	The only value is Unit (used as a placeholter)
list (t)	immutable, homogeneous linked list
pair (l) (r)	A pair of two values (a) and (b) of type (l) and (r): (Pair a b)
option (t)	Optional value of type (t): None or (Some v).
or (l) (r)	A union of two types: a value holding either of (l) or (r)
set (t)	Immutable sets of values of type (t)
map (k) (v)	Immutable maps from keys of type (k) of values of type (v)
big_map (k) (v)	Lazily deserialized maps from keys of type (k) of values of type (v)

12.4 Unfamiliar Datatypes

- **or (l) (r)**
  Technically a sum type. **Example:** Value of type or string bool can be Left ”foo” or Right False.

```
LEFT :: 'a : 'S -> or 'a 'b : 'S
RIGHT :: 'b : 'S -> or 'a 'b : 'S
IF LEFT code1 code2 :: or 'a 'b : 'S
 removes top value,
 executes code1 on ('a : 'S) if it was a Left 'a,
 otherwise executes code2 on ('b : 'S) if it was a Right 'b
```

- **set (t)**
  Immutable set with elements of type t. Elements must be comparable.

```
EMPTY SET
ITER code  :: set 'a : 'S -> 'S
apply code :: 'a : 'S -> S to each element of the set (cf. Python for)
MEM
SIZE
UPDATE :: 'a : bool : set 'a : 'S -> set  'a : 'S
 applied to (v : b : s :. . . ) return a new set (s' : . . . ) such that
 s' has the same elements as s except
 if b=True, then v in s 0
 if b=False, then v not in s'
```

- **big map (k) (v)**
  Instructions on big maps have higher gas costs than those over standard maps, as data is lazily deserialized. However, a big map has a lower storage cost than a standard map of the same size.

- **Stack Instructions**

Instruction	Description
DROP	Drop the top element of the stack
DUP	Duplicate the top element of the stack
SWAP	Exchange the top two elements of the stack
PUSH 't x	Push a value of x of type t onto the stack
UNIT	Push unit value
LAMBDA 'a 'r code	Push function given by code; arg type 'a; return type 'r
NIL t	Push an empty list of type (list (t)

- **Instructions on pairs**

Instruction	Description
PAIR	Build a pair from the stack's top two elements
CAR	Push the left part of a pair onto stack
CDR	Push the right part of a pair onto stack
COMPARE	Lexicographic comparison

- **Comparisons**

Instruction	Description
EQ	Checks that the top element of the stack is equal to zero
NEQ	Checks that the top element of the stack is not equal to zero
LT	Checks that the top element of the stack is less than zero
GT	Checks that the top element of the stack is greater than zero
LE	Checks that the top element of the stack is less than or equal to zero
GE	Checks that the top of the stack is greater than or equal to zero

- **Istructions on Strings**

Instruction	Description
CONCAT	String concatenation
SIZE	The number of characters in a string
SLICE	String access
COMPARE	Lexicographic comparison

- **Domain specific**

Data Type	Description
timestamp	Dates in real world
mutez	specific type for manipulating tokens
operation	internal operation emitted by a contract
contract	contract
address	untyped contract address
key	public cryptography key
key_hash	hash of a public cryptography key
signature	cryptographic signature

- Example:
A Tezos contract cannot manipulate the blockchain directly, it returns a list of operations to execute after the contract terminates (**metaprogramming**). Types operation are created by: CREATE_CONTRACT, SET_DELEGATE, TRANSFER_TOKEN.

12.5 TRANSFER_TOKENS

To send a specified amount of tokens to a contract.
Example: suppose the contract's address is a and it expects a parameter of type string.
TRANSFER TOKENS :: "hello" : 1000 : a : S 7 − > transfer tokens "hello" 1000 a : S
if the target contract is an implicit account, the parameter type is *unit*.

An implicit account is not associated with a specific contract. Rather, its contract is implicit, which mean it just updates the account balance with the tokens received.

Schedule Transfer to Bob

- Suppose Bob's address is "tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx" (do not use this address)
- IMPLICIT_ACCOUNT transforms a key hash into its implicit account

```
parameter unit;
storage unit;
code {
  DROP; # ignore parameter and storage
  PUSH key_hash "tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx"; # key_hash  :  A
  IMPLICIT_ACCOUNT;        # contract  unit  :  A
  AMOUNT;                  # mutez : contract unit : A
  UNIT;                 # unit : mutez : contract unit : A
  TRANSFER_TOKENS;         # transfer mutez−>contract : A
  DIP{UNIT; NIL operation} # transfer mutez−>contract : NIL : unit : A
  CONS; PAIR               # ([transfer mutez−>contract], unit)
}
```

12.6 SET_DELEGATE

Construct operation to add, update or remove a delegation (i.e., baking right).
SET DELEGATE :: option key hash : 'S − > operation : 'S
None withdraws delegation, Some *kh* delegates to kh.

12.7 CREATE_CONTRACT

CREATE CONTRACT ty1 ty2 code :: option key hash : mutez : ty1 : 'S − > operation : address : 'S

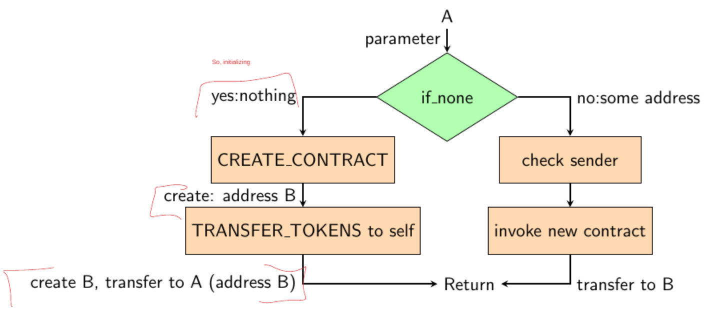
the optional *key_hash* determines a delegate, the initial value of the storage *ty1*. **Note:** On a stack machine, an operation can return more than one result!

```
parameter unit;
storage (option address);
code {
      DROP;
      UNIT;            # Initial storage for contract
      AMOUNT;             # Push initial balance
      NONE key_hash;      # No delegate
      CREATE_CONTRACT      # Create the contract
        { parameter unit ;
          storage unit ;
          code
            { CDR;
              NIL operation;
              PAIR; }
        };
      DIP {SOME;NIL operation};CONS ; PAIR   # Epilogue
}
```

```
      CREATE_CONTRACT {...}     # Create the contract
# [ cc−operation , address ]
      DIP {SOME;NIL operation};
# [ cc−operation , NIL , SOME address ]
      CONS;
# [ [cc−operation], SOME address ]
      PAIR  # Epilogue
# [ pair [cc−operation] (SOME address) ]
```

final storage: SOME *address* of the new contract to be created, **BUT: we cannot call the new contract in this list of operations**
To do so, we'd have to create a *TRANSFER TOKENS* operation with this address, but it's not yet valid.

```
parameter (option address) ;
storage unit ;
code { CAR ;
      IF_NONE
        { PUSH string "dummy" ;
          PUSH mutez 100000000 ; NONE key_hash ;
          CREATE_CONTRACT
            { parameter string ;
              storage string ;
              code { CAR ; NIL operation ; PAIR } } ;
          DIP { SOME ; DIP { SELF ; PUSH mutez 0 } ; TRANSFER_TOKENS ;
                NIL operation ; SWAP ; CONS } ;
          CONS ; UNIT ; SWAP ; PAIR }
        { SELF ; ADDRESS ; SENDER ; IFCMPNEQ { FAIL } {} ;
          CONTRACT string ; IF_NONE { FAIL } {} ;
          PUSH mutez 0 ; PUSH string "abcdefg" ; TRANSFER_TOKENS ;
          NIL operation ; SWAP ; CONS ; UNIT ; SWAP ; PAIR } } ;
```



```
        { PUSH string "dummy" ;
          PUSH mutez 100000000 ; NONE key_hash ;
          CREATE_CONTRACT
            { parameter string ;
              storage string ;
              code { CAR ; NIL operation ; PAIR } } ;
# cc−operation : address
          DIP { SOME ; DIP { SELF ; PUSH mutez 0 } ;
# cc−operation : SOME address ; mutez 0; my−address
     Exa    TRANSFER_TOKENS ;
# cc−operation : tt−operation
                NIL operation ; SWAP ; CONS } ;
# cc−operation : [tt−operation]
          CONS ; UNIT ; SWAP ; PAIR }
# pair [cc−operation, tt−operation] unit
```

```
# stack: new−address
# check if I'm the sender
        { SELF ; ADDRESS ; SENDER ; IFCMPNEQ { FAIL } {} ;
# stack: new−address
# check if this is a contract with string input
          CONTRACT string ; IF_NONE { FAIL } {} ;
# stack: contract
# schedule invocation of the contract
          PUSH mutez 0 ; PUSH string "abcdefg" ; TRANSFER_TOKENS ;
# stack: tt−operation
          NIL operation ; SWAP ; CONS ; UNIT ; SWAP ; PAIR } } ;
# stack: pair [tt−operation] unit
```

13 Smart Contracts Security

13.1 DAO - Decentralized Autonomous Organization

Est. 2016, Initial purposes were to advertise smart contracts projects and fund raising.
- How it works: Proposers advertise projects, investors vote for projects according to the size of their investment (Ether), implemented by smart contracts: collect funding and votes and the pay out according to algorithm.
Anyway, it was hacked, money was extracted before the funding phase had started, ended by an hard fork. ETH stopped all exchanged and forgot DAO.

- How it was hacked: after some delay, DAO is splitted in children. **Loophole:** retrieve Ether first, update the balance later, during it recursively call the split procedure → retrieve the funds multiple times.

- Technical background of the hack:
Ethereum contract specifies a **default or fallback** function, invoked when no function is matched by the message.

```
contract Sink {
    function() external payable { }
}
```

- Vulnerable Contract:

```
function getBalance(address user) constant returns(uint) {
  return userBalances[user];
}

function addToBalance() {
  userBalances[msg.sender] += msg.amount;
}

function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }
  userBalances[msg.sender] = 0;
}
```

- msg.sender address of sender
- msg.amount Ether that comes with the message
- .value(...) adds Ether to a method call
- what can go wrong?

- Attack

The code invokes the fallback function of the sender, the attacker can withdraw twice.

```
VulnerableContract v;
uint times = 0;
...
v.withdrawBalance();
...
function () {
  // To be called by a vulnerable
  // This will double withdraw.

  if (times == 0) {
    times = 1;
    v.withdrawBalance();

  } else { times = 0; }
}
```

- Mitigations:

Use correct ordering (requires careful code rewrite)

```
function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  userBalances[msg.sender] = 0;
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
}
```

Insert "mutex" (requires stylized code rewrite)

```
function withdrawBalance() {
  if ( withdrawMutex[msg.sender] == true) { throw; }
  withdrawMutex[msg.sender] = true;
  amountToWithdraw = userBalances[msg.sender];
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
  userBalances[msg.sender] = 0;
  withdrawMutex[msg.sender] = false;
}
```

13.2 KotE - King of the Ether

KotE is a contract living on the Ethereum blockchain, a classical ponzi scheme that makes you the king or the queen of the Blockchain.

- Rules:

To become Monarch, send a certain price (e.g. 10 ETH), the contract will send that amount (less a small fee) to the previous Monarch, the next Monarch will pay +33% to become it. If no one will pay, after 14 days the throne will decade and the price reset to 0.5 ETH, the last Monarch will not get a refund.

First version had a bug that didn't guarantee the successful payment.

```
/*** Listing 1 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  winner.send(1000); // send a prize to the winner
  prizePaidOut = True;
}
```

If send fails (out of gas, callstack overflow) the Monarch will not get his money.

From Ethereum documentation

There are some dangers in using send: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of send or even better: Use a pattern where the recipient withdraws the money.

Revised code

```
/*** Listing 2 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (winner.send(1000))
    prizePaidOut = True;
  else throw;
}
```

Works ok

Suppose we also want to give a litte to the loser

```
/*** Listing 3 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (winner.send(1000) && loser.send(10))
    prizePaidOut = True;
  else throw;
}
```

- protects from callstack overflow
- but introduces dependency between winner and loser

```
/*** Listing 4 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (callStackIsEmpty()) throw;
  winner.send(1000)
  loser.send(10)
  prizePaidOut = True;
}
```

If instead of sending tokens, the contract sets up an entrypoint where recipients can pick up their tokens.

```
/*** Listing 5 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  accounts[winner] += 1000
  accounts[loser] += 10
  prizePaidOut = True;
}
...
function withdraw(amount) {
  if (accounts[msg.sender] >= amount) {
     accounts[msg.sender] -= amount;
     msg.sender.send(amount);
  }
}
```

The "King of the Ether Throne" lottery game is the most well-known case of this bug so far. This bug wasn't noticed until after a sum of 200 Ether (worth more than USD 2000 at today's price) failed to reach a rightful lottery winner. The relevant code in King of the Ether resembles that of Listing 2. Fortunately, in this case, the contract developer was able to use an unrelated function in the contract as a "manual override" to release the stuck funds. A less scrupulous administrator could have used the same function to steal the Ether!
Almost a year earlier (while Ethereum was in its "frontier" release), a popular lottery contract, EtherPot [9], also suffered from the same bug.
An earlier version of BTCRelay also exhibited this bug [7]. Although the hazard was noticed in an earlier security audit, the wrong fix was applied at first [8].

https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/

13.3 Parity Multisig Bug

In July 2017, the amount of 153,037 Ether (worth more than USD 30M) was stolen from three large Ethereum multisig wallet contracts. The MultisigExploit-Hacker (MEH) exploited a vulnerability in the Parity 1.5 client's multisig wallet contract. MEH took ownership of a victim's wallet with a single transaction.

- Contract Libraries:

Ethereum contracts can be split into multiple parts → libraries.
Libraries can be updated without changing the address of the importing contract → flexibility, upgrades and bug fixes possible.

The Wallet Library

```
contract WalletLibrary {
    address owner;

    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
            owner = _new_owner;
        }
    }

    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
} } }
```

The Wallet Contract

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        _walletLibrary = <address of pre−deployed WalletLibrary>;
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }

    function withdraw(uint amount) returns (bool success) {
        return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")), amount);
    }

    // fallback function gets called if no other function matches call
    function () payable {
        _walletLibrary.delegatecall(msg.data);
    }
}
```

_walletLibrary is assumed to hold the address of a deployment of the WalletLibrary given a suitable method, this address can be changed.

- Delegation: Each contract maintains a table of external entry points, Indexed by SHA3 hash, the delegatecall method takes such a hash and further arguments and calls the method via this index hence, the constructor calls the initWallet method of the library. the fallback function delegates to the library by passing msg.data this forwarder enables calling any function in the library, unless the call is already handled by Wallet.

- Attack: The attacker calls initWallet with its own address on a Wallet contract As Wallet has no matching method, the fallback is invoked. The fallback delegates to WalletLibrary which changes the owner of the Wallet. Next, the attacker can withdraw everything!

- How to prevent it: Solidity can mark functions as internal or external. An internal initWallet function would not show up in the table of entry points.