Masaryk University
Faculty of Informatics

Decentralized Application Platform Ethereum

Master's Thesis

Bc. Michal Galan

Brno, Fall 2016

Masaryk University
Faculty of Informatics

Decentralized Application Platform Ethereum

Master's Thesis

Bc. Michal Galan

Brno, Fall 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Advisor: RNDr. David Sehnal Ph.D.

Acknowledgement

I would like to thank my advisor RNDr. David Sehnal Ph.D. for his guidance and patience.

Abstract

Trust in authority is a part of centralized systems such as banking, e-commerce, web search engines and social platforms.

Complex business relations and transaction fees exist between financial authorities in international credit card systems. Moreover, these trust-based payment systems can be practically affected by outside factors such as political influence.

Bitcoin is an implementation of cash system which can transfer bitcoin currency internationally without authorities and centralized servers. This thesis describes core concepts of cryptocurrencies in general and implementation details of Bitcoin – transactions, authentication mechanism, blockchain and distributed consensus. Security implications of majority attack as well as generalized brute force attack are also discussed.

Four related contract concepts emerged in the past. Contract in law, Ricardian contract defined by Ian Grigg, Smart contract defined by Nick Szabo and Smart contract used in Ethereum. Smart contracts (Frozen deposit and Escrow and dispute mediation) can be implemented in systems such as Bitcoin.

Ethereum uses similar concepts known from Bitcoin. Moreover it introduces new concepts for more practical development – accounts, messages, Ethereum Virtual Machine and gas-based fee system. We proposed simplified, yet realistic problem with a contract solution as well as smart contract implementation in Solidity programming language. The smart contract was deployed to the Ethereum blockchain.

Keywords

bitcoin, ethereum, cryptocurrency, smart contract

Contents

List of Tables

List of Figures

Property

Introduction

Satoshi Nakamoto (pseudonym) released a peer-to-peer electronic cash system called Bitcoin [1] in the year 2009. The system is the first known open-sourced digital cash system with distributed consensus algorithm and public transaction ledger. As long as the majority of Bitcoin nodes are controlled by honest actors, the system converges to a consensus of valid transactions. No central authority or central server is needed to engage in the process in contrast with traditional payment networks.

Bitcoin peer-to-peer network is formed by network nodes which publish, process, validate and propagate bitcoin tokens, the currency with limited supply. Bitcoin addresses as well as computer code can be used as authorization mechanism to control specific bitcoin tokens. The idea of decentralized code execution was later developed and implemented in Ethereum cryptocurrency [2]. Distributed consensus on data in Bitcoin as well as in Ethereum is possible due to the mining process with proof-of-work system.

As of 2016, there are more than 500 actively traded cryptocurrencies [3] which are based on Bitcoin source code or custom source code. They implement various modifications to the original Bitcoin implementation or new non-monetary use cases of distributed consensus. As an example, Litecoin is a software fork of Bitcoin with different currency issuance policy and different proof-of-work algorithm [4]. Zcash uses Zerocash protocol [5] and Monero uses CryptoNote protocol [6] to hide transactions details included in their public blockchains. Ethereum uses virtual machine to execute computer programs (smart contracts) in distributed environment by a random network node and by all network nodes as part of distributed consensus.

The idea of smart contracts was published on the Internet in 1994 by Nick Szabo [7]. Although Bitcoin was officially presented as decentralized cash system, it can be used as development platform for smart contracts. However, Ethereum offers more advanced approach to smart contract development.

The first chapter extends with examples the original ideas and motivations behind Bitcoin. The second chapter explains concepts used in cryptocurrencies and implementation details of the first cryptocurrency. The third chapter is an introduction to different concepts of contract. Finally, the fourth chapter briefly explains Ethereum and presents our idea of Household contract with smart contract implementation.

1 Decentralization

1.1 Cashless society

In the year 2015, 53% of European individuals aged 16 to 74 years had purchased goods or services over the Internet. Eight years earlier in the year 2007, this proportion was 30% [8]. The Internet enables customers to send payment transaction instantly from personal computer or mobile phone. Merchants can receive payments in days or hours from distant countries. However, not only Internet cashless payments are popular. As of August 2016, 80% of all transactions in Sweden are made by cards [9]. The popularity of Internet payments and card payments has important consequence – customers and businesses use less cash. Moreover, there is another factor which supports this trend.

Laws passed in Italy (2011), Spain (2012) and France (2015) forbid cash transactions over certain limit [10, 11] and similar law was proposed in Germany in 2016 [12]. The limits are 1000 EUR, 2500 EUR, 1000 EUR and 5000 EUR respectively. The laws in case of Italy and Spain target tax evasion, the laws in France and Germany target money laundering and financing of terrorism. In the event of inflation without limit readjustments, or if more political restrictions are applied, cash transactions could become impractical to use.

As a result of these factors, a traditional push transport of cash between two parties – a sender and a recipient, is being replaced by an electronic pull transfer of information bits between three parties – a sender, a mediator and a recipient.

1.2 Trust based payment systems

Electronic financial transactions of private currencies (US Dollar, Euro or Czech koruna) between a sending party and a receiving party are processed, validated and confirmed by one or more mediator parties.

An example with one mediator party is a payment from bank account A to another bank account B in the same bank. The bank verifies if the account A has enough balance and coordinates own bookkeeping process to preserve atomicity of addition and subtraction of those two account balances.

Another example involves two mediators – a person with a bank account makes a payment to another person in a different bank. Both banks participate in coordination of the payment. However, in practice, communication between two banks can be mediated by yet another mediator – SWIFT network [13], Visa or Mastercard company. In international card payments, even more than three mediators can participate in electronic currency transfer as figure 1.1 shows.

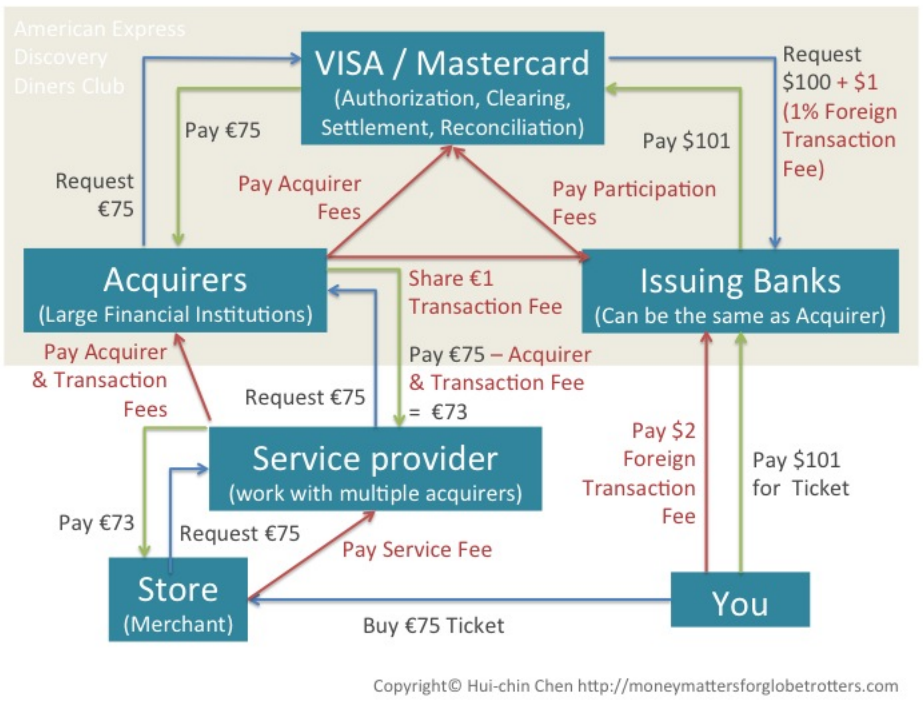
American Express
Discovery
Diners Club
VISA / Mastercard
(Authorization, Clearing, Settlement, Reconciliation)
Request $100 + $1 (1% Foreign Transaction Fee)
Pay €75
Pay $101
Request €75
Pay Acquirer Fees
Pay Participation Fees
Acquirers
(Large Financial Institutions)
Share €1 Transaction Fee
Issuing Banks
(Can be the same as Acquirer)
Pay Acquirer & Transaction Fees
Request €75
Pay €75 – Acquirer & Transaction Fee = €73
Service provider
(work with multiple acquirers)
Pay $2 Foreign Transaction Fee
Pay $101 for Ticket
Pay €73
Request €75
Pay Service Fee
Store
(Merchant)
Buy €75 Ticket
You
Copyright© Hui-chin Chen http://moneymattersforglobetrotters.com

Figure 1.1: Diagram of international transaction with card [14]

The figure 1.1 shows a fictional 75€ ticket payment. The ticket is being bought from a merchant in an European country via Visa (or Mastercard) payment card issued in the United States. The buyer (you) pay Foreign transaction fee to your issuing bank, the issuing bank pays Participation fee to Visa (or Mastercard), the Acquirers pay Acquirer fees to Visa (or Mastercard), merchant's Service provider pays Acquirer and Transaction fees to the Acquirer and finally, the merchant pays Service fee to the Service provider. In domestic payments, the count of mediators can be lower if Acquirer and Issuing bank are identical institutions [14].

The mediators have an economic incentive to generate profit. However, the profit is composed of transaction fees paid by the buyer and the merchant, or from other business activities. Business cartels and monopoly can be formed if new mediators are not being created.

These payment systems contain component of trust in authority. The buyer has to trust each mediator in the chain of mediators that the payment will be delivered to the merchant. Similarly, the merchant has to trust the chain of mediators that it will deliver the confirmation message back to the buyer.

However, trust-based digital payment systems may not be reliable. Politics, business strategy or personal motivation can create pressure on providers of trust-based systems to change behaviour and discriminate various users. In 2010, after the launch of Cablegate by Wikileaks, companies Bank of America, VISA, MasterCard, PayPal and Western Union imposed financial blockade on Wikileaks, which resulted in elimination of 95% of its revenue [15].

Monopolization as well as the component of trust in authority are nowadays present in many other non-financial Internet-based systems such as as web searching services, e-commerce platforms, social platforms and communication platforms.

2 Concepts and mechanics – Bitcoin

In this chapter we describe main Bitcoin concepts and implementation details. As the first decentralized cryptocurrency and a currency with the highest market capitalization [3] as of December 2016, it is probably the most documented one. The understanding of core concepts and implementation details of Bitcoins is beneficial for further study of other cryptocurrencies such as Ethereum.

2.1 Network node roles

Bitcoin uses peer-to-peer (P2P) distributed network architecture. Network nodes communicate via TCP protocol with default port 8333 (chapter 6 in [16]). Generally, there are four roles critical to secure and use Bitcoin system. The roles are not mutually exclusive – a node can, or needs to support functionality of multiple roles. The summarized roles are further explained in following sections of this thesis.

2.1.1 Routers

Nodes with the routing functionality help other nodes to discover new nodes in a network discovery process. They implement a middleware which allows to receive and relay Bitcoin transactions. Most of Bitcoin software has implemented router functionality.

2.1.2 Wallets

Nodes with the wallet functionality allow to create and relay new Bitcoin transactions and track past payments. They can support additional functionality – for example, cryptographic signing of text. Wallet functionality is included in Bitcoin software such as Bitcoin Core, Armory or Electrum [17].

2.1.3 Miners

Nodes with the mining functionality receive and pack new transactions into data structures called blocks. Miners compete between each other in solving of computationally hard problem. In effect, they create network-wise distributed consensus. As the first source of reward they collect transaction fees specified by transaction creators or wallets. As the second source of reward they create and collect new bitcoins. Although Bitcoin Core has implemented CPU mining functionality, mining with CPU on a personal computer is not profitable in the year 2016. CGminer [18] and BFGminer [19] support mining with ASIC and FPGA hardware.

2.1.4 Full nodes

Nodes labeled as full blockchain nodes or full nodes [20] maintain complete database of all Bitcoin transactions. They can anonymously validate any existing transaction [21]. The size of blockchain in 2016 increased approximately from 50 GB to 100 GB [22]. Bitcoin Core, Armory, mSIGNA and mining software are examples of full nodes.

2.2 Addresses

Email become popular method for information exchange and "email address" is well-known concept for email users. Format of email address originally consisted of user identifier and host name, separated by "at" sign (example: Postel@ISIF). Later it was changed to include domain names to be compatible with Domain Name System [23]. Similarly, "Bitcoin address" is well-known concept for Bitcoin users and it is often used as destination identifier when sending bitcoins. However, Bitcoin address does not directly identify particular user, machine, network location, nor sub-network. In fact, it is not a location identifier. In this section we describe addresses in more detail.

2.2.1 Address construction

Bitcoin addresses are often presented to users as alphanumeric strings [24]. We generated five Bitcoin addresses in bitaddress.org, an open source JavaScript Bitcoin address generator:

```
1Ag71eTr2TjtbojAkBfmVFkmGWC2Z1VPC8
1PXHCkCMS1BAFXWAUHfoUnVKkg2JREyB3h
```

```
1GgmytEkDRLEJffimupMv7B1mrvN9QHYH8
19PCeT6Y9gRGbq5cVB6uu7B7h7zAT11khZ
12imR6JKttY76RwrMmkrJZrCnKZ5zCLTYM
```

The fact that all five generated addresses start with the character "1" is not a coincidence. To understand what they represent we describe parts of their generation process (chapter 4 in [16]):

```
HASH_RAW = RIPEMD160(SHA256(DATA))
ADDRESS = BASE58CHECK(VERSION_BYTE+HASH_RAW)
```

where RIPEMD160 is cryptographic hash function [25], SHA256 is cryptographic hash function [26] and BASE58CHECK is encoding function with built-in checksum function [27]. VERSION_BYTE is parameter containing information about type (version) of DATA parameter. Finally and most importantly, the DATA parameter contains data which constructs Bitcoin address (chapter 4 in [16]).

The VERSION_BYTE is concatenated with the HASH_RAW before being encoded. This explains why all the generated addresses by bitaddress.org start with symbol "1" – they all have the same VERSION_BYTE.

2.2.2 Address types

As of 2016 there exist multiple address types (versions) with specific leading characters (chapter 4 in [16]):

- P2PKH address (Pay-to-PubkeyHash, starts with "1")
- P2SH address (Pay-to-ScriptHash, starts with "3")
- P2PKH testnet address (Pay-to-PubkeyHash, starts with "m"/"n")
- P2SH testnet address (Pay-to-ScriptHash, starts with "2")

The DATA parameter can contain information of two types:

- ECDSA public key (in P2PKH mainnet or testnet addresses)
- Script program (in P2SH mainnet or testnet addresses)

Bitcoin address does not represent a location. Rather it represents either a hashed public key or a hashed computer program. In the following section we describe Bitcoin transactions.

2.3 Transactions

Bitcoin transactions represent transfer of value. Everybody in the peer-to-peer Bitcoin network can construct new transaction and send it into the network through some P2P node.

The transaction is verified by the node when received. It must have correct syntax, correct byte size, value no less than zero, the bitcoins inside the transaction must be authorized to be spend and it must satisfy other protocol rules [28, 29]. When the transaction is successfully verified, the node propagates the transaction to other known nodes. This verification and propagation process repeats itself on other nodes and with other transactions. In effect, transactions are discovered by all Bitcoin participants in couple of seconds.

However, it is not sufficient to verify new transactions by equal Bitcoin nodes. Someone could double-spend the same bitcoins or just propagate the same transaction twice. Consensus on validity of transactions as well as consensus on one transaction history must be achieved.

In Bitcoin, no central authority validates transactions and prevents malicious attempts such as double-spending of value. No central authority decides which transactions are or are not permitted. Instead, distributed consensus is reached by a process called mining.

2.3.1 Transaction structure

Bitcoin transaction is a data structure which encodes information about transfer of bitcoins. It consists of multiple data fields with various byte size as shown in table 2.1.

Size	Field
4 bytes	Version
1-9 bytes	Input counter
Variable	Inputs
1-9 bytes	Output counter
Variable	Outputs
4 bytes	Locktime

Table 2.1: Transaction data structure (chapter 5 in [16])

Version field is included for backward compatibility if the transaction format will change in the future.

Input counter and Output counter store number of Inputs and Outputs. We describe inputs and outputs later. They are important concepts as well as structures which implement Bitcoin value transfer and Bitcoin authentication mechanism.

Locktime contains unix timestamp or block height (logical time) and specifies the earliest time when a transaction should be included in Bitcoin transaction ledger, the Bitcoin blockchain.

2.3.2 Inputs and Outputs

Inputs and outputs are both fundamental concepts and data structures in Bitcoin. They represent a transfer of value (bitcoin tokens) and they store data needed for authorization to spend (move) bitcoins.

As figure 2.1 shows, the value transfer is realized by connecting an output to an input in newer transaction. Mixing of input values is used to virtually split value into one or multiple outputs.

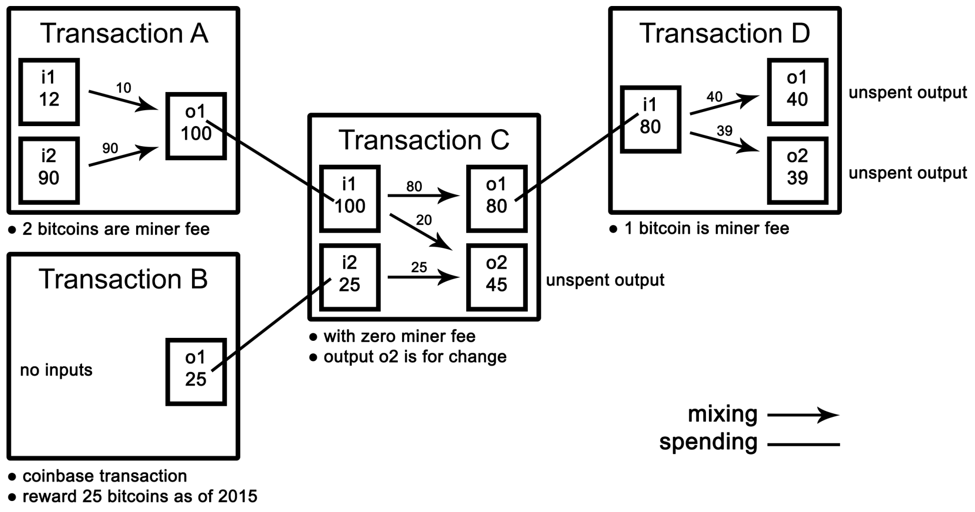

Figure 2.1: Spending and mixing of value (bitcoin unit)

Output

An output represents and contains bitcoin value in satoshi unit (1 bitcoin = 10^8 satoshi) as table 2.2 shows. In addition, each output contains authorization conditions encoded in computer program called locking script. The conditions are needed to be met to spend the output.

Size	Field
8 bytes	Value (satoshi)
1-9 bytes	Locking script size
Variable size bytes	Locking script

Table 2.2: Transaction output (chapter 5 in [16])

Input

An input represents a reference to an output in some existing transaction. The reference represents transfer of value between two transactions – the spending. The act of an output spending makes the output unspendable. However, the value of referenced output is virtually moved to a new transaction where the value can be mixed or split into new outputs. Similarly to locking scripts in outputs, an input contains computer program called unlocking script. An unlocking script must satisfy conditions set by a locking script in a referenced output.

Transaction fee and coinbase transaction

As figure 2.1 shows, if the sum of all outputs in one transaction is less than the sum of all inputs in the same transaction then the difference is a transaction fee accounted to the miner who inserts the transaction into Bitcoin blockchain.

There exists a special coinbase transactions with input of zero value and one output. A coinbase transaction emits new bitcoins into the system. It can be created only by bitcoin miners as a reward for creating the distributed consensus.

Size	Field
32 bytes	Previous transaction hash
4 bytes	Output index
1-9 bytes	Unlocking script size
Variable size bytes	Unlocking script
4 bytes	Sequence number

Table 2.3: Transaction input (chapter 5 in [16])

2.3.3 Script

Script is an imperative, stack-based programming language [30] used in locking and unlocking scripts. It is not Turing-complete – it does not support loops and conditional jumps. This property protects Bitcoin software from infinite loops [31] or malicious attacks targeting performance of Bitcoin hardware. In addition, Script programs are stateless (chapter 5 in [16]) – all data needed by a script program must be included in the program and no data can be stored outside the program – performance of Script programs does not depend on performance of input and output operations.

Word	Opcode	Input	Output
OP_FALSE	0	-	0
OP_TRUE	81	-	1
OP_NOP	97	-	-
OP_IF	99	<expression>	-
OP_VERIFY	105	true / false	- / fail
OP_RETURN	106	-	fail
OP_TOALTSTACK	107	x1	(alt stack) x1
OP_FROMALTSTACK	108	(alt stack) x1	x1
OP_SWAP	124	x1 x2	x2 x1
OP_EQUAL	135	x1 x2	true / false
OP_ADD	147	a b	c
OP_MAX	164	a b	c

Table 2.4: Subset of Scripts words [30]

The following simple C or Java program is written in table 2.5 in Script language.

```
((2 + 5) == 8);
```

The program is written and executed in left-to-right direction and contains three literals (numbers) and two commands (opcodes).

Step 1.	Step 2.	Step 3.	Step 4.	Step 5.
2	5	OP_ADD	8	OP_EQUAL

Table 2.5: Script program example

1. Number 2 is pushed to the main stack: `|2|`.
2. Number 5 is pushed to the main stack: `|2,5|`.
3. The two last items are popped from the stack and their sum is pushed to the stack: `|7|`.
4. Number 8 is pushed to the main stack: `|7,8|`.
5. The two last items are popped from the stack and compared for equality – they are not equal therefore FALSE (0) is pushed to the stack: `|0|`.

When a script program terminates, the item at the top of the main stack is the result value of the program.

2.3.4 Locking and unlocking scripts

Transfer of value in Bitcoin is implemented by inputs and outputs as we described in previous sections. This section describes the authorization mechanism of Bitcoin transactions.

A transaction output contains computer program called locking script. The program defines if and how the output can be spend. Similarly, each input contains computer program called unlocking script. Unlocking script is a proof to the corresponding locking script that the output can be spend. Locking and unlocking scripts together implement the authorization mechanism of value transfer.

Bitcoin full nodes execute the following procedure to verify the authorization to spend a transaction.

- Input I with referenced output O is selected from the transaction.

- The unlocking script of input I is executed with a stack S.
- If the unlocking script program finishes without errors then the locking script of output O is executed with leftover stack S content.
- If the locking script program finishes without errors and the result value on the stack S is TRUE then the input I is authorized to spend the output O.
- In other cases (if there is an error during execution of any script or the resulting value is not TRUE) the procedure finishes and the transaction is considered as not valid.
- Another input with referenced output is selected and the procedure repeats itself while there is at least one unchecked input and the transaction is considered as valid.

Locking script is also called Pubkey Script (ScriptPubKey) in Bitcoin knowledge sources [32]. Similarly, Unlocking script is also called Signature script (ScriptSig) [33].

The reason is historical – early versions of Bitcoin software were "locking" outputs by including a public key into locking scripts. An input which would spend the output had to prove the knowledge of the corresponding private key by including a signature. The authorization mechanism verified the public key and the signature and "unlock" the output if the signature was correct. As of 2016 this authorization scheme is obsolete [34] but the names are still in use.

We can define user balance as the sum of unspent outputs which are spendable by an user. Indeed, Bitcoin wallets calculate balance in this manner. Similarly, the sum of all unspent Bitcoin outputs ever created is equal to all bitcoin value in existence.

2.3.5 Standard scripts and transactions

Bitcoin Core software defines five types of transactions which are called Standard transactions [35]. A transaction is standard if it passes the IsStandard test, otherwise it is non-standard [36]. Standard transactions are commonly used in Bitcoin network and in the future the distinction from non-standard transactions can be modified or lifted [37]. Other Bitcoin client software and miners may not differentiate between standard and non-standard transactions (chapter 5 in [16]).

Standard transactions allows standard scripts (authorization schemes) listed below. One standard transaction can contain multiple outputs with different standard scripts. However, if a transaction contains at least one non-standard script, the transaction is non-standard.

1. Pay-to-Public-Key (P2PK)

```
LS:   <public_key> OP_CHECKSIG
ULS:  <signature>
```

Pay-to-Public-Key (P2PK) (chapter 5 in [16]) is the authorization scheme discussed in previous section. Firstly, the unlocking script is executed – signature is added to the main stack. Locking script is then executed – public key is added to the stack and OP_CHECKSIG is called. OP_CHECKSIG [38] takes the two top stack items and checks if the signature was created by a private key corresponding to the public key. If and only if the signature is correct the input is authorized to spend the output. In other words, only owner of the public key stored in output can spend the output.

2. Pay-to-Public-Key-Hash (P2PKH)

```
LS:   OP_DUP OP_HASH160 <public_key_hash>
      OP_EQUAL OP_CHECKSIG
ULS:  <signature> <public_key>
```

Pay-to-Public-Key-Hash (P2PKH) (chapter 5 in [16]) is functionally similar to Pay-to-Public-Key. Firstly, the unlocking script is executed – signature and corresponding public key from input are pushed to the stack. Then OP_DUP duplicates the public key and OP_HASH160 hashes the public key duplicate. Then the hash of the public key from output is pushed to the stack and OP_EQUAL checks if the hash from the output and the hashed public key from the input are equal.

The remaining program execution is the same as Pay-to-Public-Key in which OP_CHECKSIG checks signature with public key. This authentication scheme is commonly used and it can be also called as Pay-to-Address. The <public_key_hash> stored in output is Bitcoin address without BASE58CHECK encoding and without the version byte (first symbol of Bitcoin address).

One advantage of Pay-to-Public-Key-Hash authentication scheme over Pay-to-Public-Key is the absence of unprotected public key in outputs. A quantum computer with implemented Shor's algorithm could perform an attack on elliptic curve public keys [39] in all Pay-to-Public-Key unspent outputs ever created. Attacker then could spend bitcoins from the outputs.

Quantum computer could also perform an attack on hash functions with Grover's algorithm but such attach would be less efficient [40].

However, when new transactions with Pay-to-Public-Key-Hash and public_key in inputs are propagated, nodes could still perform the quantum attack with Shor's algorithm. Lamport signatures were proposed to secure Bitcoin from quantum attacks [40].

3. Multi-Signature (Multisig)

```
LS:  <M> <public_key_1> ... <public_key_N>
     <N> OP_CHECKMULTISIG
ULS: <signature_i> ... <signature_j>
```

Multi-signature (chapter 5 in [16]) outputs are locked with N public keys. It is required to provide M signatures. Any combination of M signatures is accepted.

M-of-N multisignature outputs can implement n-factor authorization to spend bitcoins with N electronic devices such as laptop and smartphone. More importantly, it can be used by centralized services such as currency exchanges to prevent theft if critical vulnerability is exploited.

4. Pay-to-Script-Hash (P2SH)

```
LS:  OP_HASH160 <redeem_script_hash> OP_EQUAL
ULS: <inner_unlocking_script> "<redeem_script>"
```

Pay-to-Script-Hash [41, 42] is a wrapper scheme around other script schemes such as Multi-Signature (Multisig). Multisig or other schemes can be transformed into P2SH in such way that the original locking script hash is placed between OP_HASH160 and OP_EQUAL (it is then called redeem script hash). The original unlocking script and the original locking script are placed into the P2SH unlocking script as inner unlocking script and *serialized* redeem script, respectively. However, the execution process differs slightly from other schemes. Gavin Andresen concludes:

> Recognizing one 'special' form of scriptPubKey and performing extra validation when it is detected is ugly. However, the consensus is that the alternatives are either uglier, are more complex to implement, and/or expand the power of the expression language in dangerous ways. [41]

Firstly, the unlocking script is executed – <inner_unlocking_script> is executed and serialized <redeem_script> is pushed to the stack. Then OP_HASH160 pushes the hashed <redeem_script> to the stack and <redeem_script_hash> is pushed on the stack too. OP_EQUAL then compares the computed hash with the pushed hash.

The execution process differs from this step. The serialized redeem script is automatically popped from the stack and then deserialized. At this point in time, the stack still contains result of the inner unlocking script. The deserialized redeem script is executed using the stack as if it was an unlocking script.

P2SH scheme has various advantages. Consider two users exist, a sender and a receiver. The sender creates new transaction; spends input(s) of a past transaction and creates new output(s) in the new transaction. Each output must contain a locking script.

- If the recipient wants to gain ownership of sender's bitcoins by a Pay-to-Public-Key script then recipient's public key must be known to the sender in advance.
- If the recipient wants to gain ownership of sender's bitcoins by a Pay-to-Public-Key-Hash script then a hash of recipient's public key (encoded as Bitcoin address with '1' prefix) must be know to the sender in advance.
- If the recipient wants to gain ownership of bitcoins by a Multi-Signature script then a list of N public keys must be known to the sender in advance.
- If the recipient wants to gain ownership of bitcoins by a non-standard script then the locking script part of the script must be known to the sender in advance.

With Pay-to-Script-Hash, the sender needs to know only hash of redeem script (encoded as Bitcoin address with '3' prefix). The hashed redeem script has always the same length and can function as a universal payment request – the hashed redeem script can be any standard locking script. Moreover, the sender or Bitcoin network nodes do not need to know how the output will be spent in the future.

With Pay-to-Script-Hash, Bitcoin address is not only a concept of sending money to a recipient, rather it is a concept of programmable money where a recipient decides what can be done with the money.

5. Data Output (OP_RETURN)

```
    LS: OP_RETURN <data>
```

Locking scripts and transactions which contain operator OP_RETURN (chapter 5 in [16]) are unspendable. The OP_RETURN halts a script execution and makes the wrapping transaction not valid.

However, a locking script with OP_RETURN operator can contain arbitrary data of size up to 40 bytes (chapter 5 in [16]). This is a sufficient size to store cryptographic digests such as SHA256 [26, 43].

A notarization service proofofexistence.com uses outputs with OP_RETURN operator and the 40 byte space as a container for file digests. While Bitcoin consensus on one transaction history exists, a file digest stored in a transaction proves the existence of the file at the time when the wrapping block of the transaction was included on the blockchain – we present more on this topic in the next sections.

However, no centralized web service is needed to store such document digests. Any transaction can be propagated directly to the Bitcoin peer-to-peer network.

2.4 Blockchain

New bitcoin transactions are collected by miner nodes and then stored in data structures called blocks. Miners send blocks with the included transactions to other network nodes. Blocks form blockchain – linked structure consisting of blocks. Blockchain stores all outputs and inputs created to a date. It is the data on which a distributed consensus must be reached.

2.4.1 Block structure

As table 2.6 shows, a block contains block size data field, block header with metadata and list of transactions.

Size	Field
4 bytes	Block size
80 bytes	Block header
1 - 9 bytes	Transaction counter
Variable size	Transactions (outputs and inputs)

Table 2.6: Block structure (chapter 7 in [16])

A block header contains meta-data needed for data consistency checks and fields used in Bitcoin mining.

Field	Description
4 bytes	Version
32 bytes	Hash of previous block header
32 bytes	Merkle root
4 bytes	Timestamp
4 bytes	Difficulty target
4 bytes	Nonce

Table 2.7: Block header structure [44]

A block is uniquely identified by a cryptographic digest (double SHA256) of its block header (chapter 7 in [16]). As table 2.7 shows, a block header contains hash of the previous block header as a parent reference. The first block, Genesis block [45] is hard-coded into Bitcoin software and do not link to any other block – the field contains 0-bits (chapter 7 in [16]).

2.4.2 Chain of blocks

Each block is linked to other blocks by referencing a parent block. If any content of header is modified then its child block header contains invalid hash. Transaction modification is also detectable – a block header contains a Merkle root of a Merkle tree structure [46] with transactions represented as leafs. A Merkle root included in block header functions as a fingerprint of transactions in the same block.

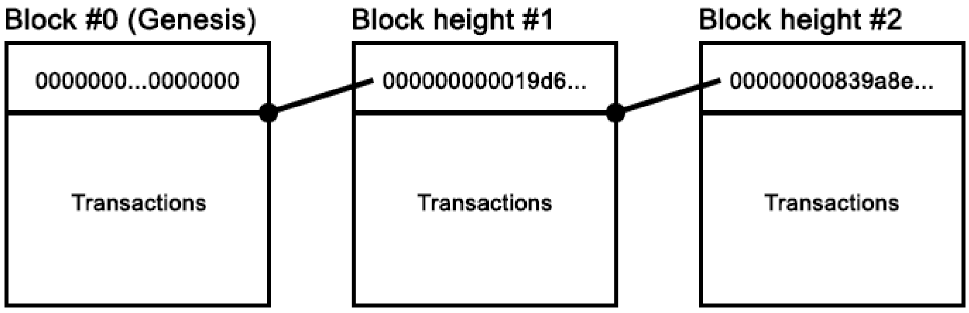

Figure 2.2: Blocks with partial hashes of parent headers [47, 48, 49]

If a transaction in the blockchain is altered, removed or added to the block at a later time then the Merkle root in the wrapping block is detectably incorrect. When a Merkle root is changed then the hash of its wrapping header differs from the hash-of-previous-block-header stored in child block. This further invalidates correctness of the next hash-of-previous-block-header field in grandchild. In fact, a change in one transaction invalidates correctness of all descendant blocks.

The probability of occurrence of non-unique block header identifier with cryptographic hash function SHA256 within sequence of first `N>1` blocks is less or equal to the probability of finding SHA256 collision with `N-1` computations in prepared setup and with prepared data. Public keys and other generated data contribute with a data entropy to their block headers therefore it can be assumed that a successful collision attack will never occur.

Figure 2.2 shows the first three Bitcoin blocks. It is not a coincidence that the hashes of the first and the second block start with multiple zeros. We explain this in the next section.

2.5 Distributed consensus

In centralized architectures such as server-client architecture, clients trust a server to provide valid information. To protect data validity, additional measures such as integrity checks or audits can be implemented on the server side. However, this creates chains of authorities where the top authorities are not be supervised and must be trusted.

In decentralized architectures such as Bitcoin, nodes do not trust other nodes and received data is considered corrupted or fraudulent. However, the definition of fraudulent data is problematic in Bitcoin because there are no network authorities to define what is and what is not a fraudulent data.

Instead of rules set by authorities, Bitcoin uses consensus rules [50]. If a node does not follow consensus rules then it is not part of the Bitcoin network and its blockchain data are not consistent with the blockchain data of other nodes [51]. For example, if a mining node creates more reward bitcoins then is allowed to by a consensus rule then the bitcoins (coinbase input) can not be stored in the blockchain. However, nodes with the same change in consensus rules can maintain and build alternative and incompatible (hard-forked) blockchains [52]. For example, other nodes can agree that the bitcoins (coinbase input) are in acceptable range.

The word consensus is used in Bitcoin literature as more generalized and broader consensus, often referenced as emergent consensus as Andreas M. Antonopoulos writes:

> Satoshi Nakamoto's main invention is the decentralized mechanism for emergent consensus. Emergent, because consensus is not achieved explicitly—there is no election or fixed moment when consensus occurs. Instead, consensus is an emergent artifact of the asynchronous interaction of thousands of independent nodes, all following simple rules. All the properties of bitcoin, including currency, transactions, payments, and the security model that does not depend on central authority or trust, derive from this invention. (chapter 8 in [16])

2.5.1 Transaction verification

In the section Transactions we briefly mentioned that Bitcoin nodes use protocol rules [28, 29] to verify new incoming transactions. The rules are part of the emergent consensus. However, some rules are included to prevent denial-of-service attacks and some can be relaxed in the future (chapter 8 in [16]).

A small subset of the rules includes:

- Transaction is syntactically correct
- Size in bytes <= MAX_BLOCK_SIZE
- Transaction is not coinbase transaction
- Transaction (TXID) is not included in transaction pool
- Transaction (TXID) is not included in blockchain
- Sum of input values >= sum of output values

2.5.2 Mining

Bitcoin nodes continuously collect, verify and propagate new incoming transactions. Transactions are collected in transaction pool (memory pool) locally on router nodes (chapter 8 in [16]).

Mining nodes (miners) are involved in additional process called mining. Mining process builds the blockchain block-by-block and it is consensually accepted by all nodes as valid.

Let us suppose a mining node which stores new transactions into a candidate block. Candidate block is a block temporarily stored in miner's memory for purpose of further actions. Height number of a candidate block is N+1 where N is the height of the highest block known to the miner which is includes in the blockchain. In other words, the candidate block is a potential child of the block with the height N. We suppose the blockchain is valid and only one block with the height N exists.

Secondly, let us assume that while the candidate block is temporarily stored and while the procedure described further in this section are in progress, a new block arrives from another miner. The block is verified [53] and stored in miner memory as the most recent block of the blockchain. The miner computes intersection of transactions included in its transaction pool and in the received block. The resulting set of transactions are transactions already included by the other miner and can be removed from the transaction pool. Then a new candidate block is constructed – it will not include any old transaction from the set. Moreover, it will be a potential child of the received block.

Transaction selection

Some transactions are more prioritized to be included in the candidate block than other. The function (chapter 8 in [16]) of transaction priority is:

$$transaction\_priority = \frac{SUM(value\_of\_input * input\_age)}{transaction\_size} \tag{2.1}$$

The value_of_input is value of bitcoins in satoshi unit (1 bitcoin = 10^8 satoshi). The input_age is the age of a referenced output by the input – the number of blocks received since the output was stored into the blockchain. The transaction_size is the size of a transaction in bytes.

$$high\_priority > \frac{100000000 * 144}{250} = 57600000 \tag{2.2}$$

The first 50 kilobytes of a transaction space are allocated for transactions with high priority (priority greater than 57 600 000) [54]. The high_priority is a threshold number which represents a transaction with one bitcoin (100000000 satoshi) aged 144 blocks (approximately 24 hours) and included in a 250 byte transaction. The reasoning used here is that if a transaction is very valuable or it waits too long to be included then it is prioritized. However, if a transaction has huge byte size than it is not prioritized.

The miner fills the remaining space of MAX_BLOCK_SIZE with other transactions. As of 2016, the constant is 1MB [55]. Miners usually include transactions with the highest miner fees first to maximize profits.

Bitcoin software clients automatically include a minimum static fee [56] or they calculate an optimal transaction fee based on past confirmation times [57].

Reward computation

Miners include one additional (coinbase) transaction in their candidate block. Coinbase transaction contains one transaction output usually with Pay-to-Public-Key or Pay-to-Public-Key-Hash scripts with miner's address. The coinbase reward (2.3) is based on the block with the highest height N included in the blockchain. Height N can be interpreted as logical time in Bitcoin, moreover it can be approximated into physical time.

$$coinbase\_reward = calculateReward(N) \tag{2.3}$$

The reward was originally 50 bitcoins, however every 210000 blocks (approximately 4 years) the reward is reduced by 50%. Such event is known as block reward halving. With this controlled supply, there will be no more than 21 million bitcoins in existence [58].

Secondly, miners calculate their miner fee as equation (2.4) shows. Miner fee will be more significant reward in the future when the block reward is reduced significantly.

$$miner\_fees = SUM(included\_inputs) - SUM(included\_outputs) \tag{2.4}$$

Finally, the total reward (2.5) for a miner is sum of miner fees and the coinbase reward.

$$total\_reward = miner\_fee + coinbase\_reward \tag{2.5}$$

The total reward is included by a miner in a coinbase transaction.

Block finalization

When a miner stores a coinbase transaction and a subset of collected transactions into a candidate block, data fields in its header (2.8) needs to be updated.

Field	Description
4 bytes	Version
32 bytes	Hash of previous block header
32 bytes	Merkle root
4 bytes	Timestamp
4 bytes	Difficulty target
4 bytes	Nonce

Table 2.8: Block header structure [44]

Version of block is set to current version. Cryptographic hash of header in block with height N is computed and stored to the candidate header.

Merkle root (cryptographic digest of included transactions) is computed and updated. Unix timestamp is updated. Difficulty target is initialized to a value measured every 2016 blocks by each miner. It is an important parameter in proof-of-work system as we describe later. Nonce field is set to zero.

Transaction selection, reward computation and block finalization are procedures run not only by one miner node but rather by all miner nodes concurrently. However, the blockchain is build block-by-block and new blocks must be serialized.

Proof-of-work system is used to set the order of blocks as well as prevent problems such as network congestion and cheap spam blocks.

Proof-of-work

Let us suppose an honest authority which randomly choose one miner of all miners every ten minutes. The chosen miner can propagate its candidate block to other nodes in the network. This approach would solve the network congestion problem. However, Bitcoin does not use such authority. Instead it uses proof-of-work [59] system Hashcash algorithm [60].

Proof-of-work algorithm used by Bitcoin miners can be expressed by a generalized pseudo code. The presented function accepts a block header and a target. It returns the potentially modified block header. In infinite loop it is changing the nonce field in a block unless a correct cryptographic hash [61] of block is found and then the nonce value is returned (wrapped in the block header).

```
block_header proofOfWork(block_header, target) {
    while (true) {
        hash = cryptographic_hash(block);
        if (hashReachedTarget(hash, target)) {
            return block_header;
        }
        block_header.nonce++;
    }
}
```

One possible implementation of the hasReachedTarget function is:

```
boolean hashReachedTarget(hash, target) {
    return hash == target;
}
```

The probability of reaching a target with the first hashReachedTarget implementation is equivalent to the probability of finding hash collision of the cryptographic_hash function [61, 62]. However, Bitcoin implementation of hashReachedTarget is different and allows partially equal hashes.

Bitcoin uses double SHA256 function SHA256(SHA256()) as cryptographic hash function. This function returns a 256 bit long digests. The target is also a 256 bit long number in Bitcoin (chapter 8 in [16]). A hash reaches a target if and only if hash is less than the target. The target parametrizes the average number of hash operations needed to find a correct hash.

Each Bitcoin miner independently recomputes optimal target on each 2016-th block. The result is called difficulty and it is encoded and represented in different format than the target. Difficulty represents target for next 2016 blocks from its computation.

$$difficulty = old\_difficulty \times \frac{minutes\_between\_blocks(N, N-2016)}{2016 \times 10} \tag{2.6}$$

The difficulty equation (2.6) measures minutes between the last 2016 blocks and adjust the old difficulty to represent a target with 10 minute average time to find correct block. The adjustments is intentionally designed to produce moving average [63]. If each miner node updated hardware to be 2-times faster at a block which is 1008 blocks after the last re-computation, the new difficulty would be only 50% greater.

2.5.3 Independent block validation

When a miner finds the correct nonce, it immediately propagates the candidate block to other nodes. Each node perform checks [53] to validate the newly received block. If it is valid then it is chained into

the local copy of blockchain and propagated to other nodes in the network.
A small subset of checks includes:

- Block is syntactically correct
- Hash of block header is less than target
- Timestamp is less than (current time + 2 hours)
- The first transaction is coinbase transaction
- Only one coinbase transaction is included
- All included transactions are valid

2.5.4 Independent blockchain selection

Although the average block time is maintained to be 10 minutes, two miners can found and start to propagate two valid candidate blocks almost at the same time and tens of milliseconds after a parent block was found. Network connection problems and network routes with different number of nodes can delay or change order of blocks.

Moreover, all miners must reach a consensus on one valid chain of blocks. Miners store in memory multiple chains of blocks. Main chain, secondary chains and orphan blocks.

Chain of blocks or chain is sequence of at least two blocks where each block has at most one parent and at most one child.

Best chain is a valid chain known to a node which has the most cumulative difficulty of blocks (greatest sum of difficulties of each block).

Main chain is a chain with candidate block being minted. It is best chain unless new better chain is received from other miners. If there are two or more chains with maximum cumulative difficulty then the best chain is the first seen chain.

Secondary chains are forks on the main chain know to a node. If block P has three sibling children A, B and C then the best chain is the chain which contains one block of (A, B, C) on the chain with the greatest cumulative difficulty. Chains containing other siblings are secondary chains. However, only one secondary chain appears in practice. And one secondary chain appears often.

Orphan block has unknown parent to a node. If orphan block has a child then they form another chain – most likely secondary chain.

Switching chains

Let us consider network of N >= 10 mining nodes. We randomly assign floor(N/2) nodes into group A and other nodes into group B.

Each miner knows the same block P which is the last block of their main chains. Each miner is mining its own candidate block with parent P. Miner Alice from group A and miner Bob from group B find valid blocks (hashes) almost at the same time. The blocks (hashes) can be different because they can contain different set of transactions, however they are both valid. Alice and Bob propagate their candidate blocks to nodes.

Let us suppose that miners in group A receive block from Alice first and miners in group B receive block from Bob first. All miners include its first received block into local copy of blockchain and start proof-of-work algorithm. When the alternative block arrives, it splits local blockchain into two secondary chains. However, heights of the first received block and alternative block are equal and if cumulative difficulties of the two chains are equal then miner can choose which chain to mine on. Restarting mining process with a new block would consume a few processing resources therefore miner continue with proof-of-work with the first block. Suppose miner Adam from group A finds valid block and propagates it to all N nodes. Each node extends the chain with Alice block. The chain with Adam's block is best chain therefore each node restarts proof-of-work algorithm with Adam's block. The event of switching to different chain is known as reconvergence.

However, Bob's block is discarded by miners. If Bob's block contained a transaction not included in Alice's block then the transaction is effectively temporarily discarded until it gets into different block again.

2.5.5 Security implications

If the blockchain reconverges to different chain with potentially different transactions, honest miners can include transactions from the invalid chain to transaction pool. However, malicious miners can exist in the network too.

Bitcoin transactions use concept of *confirmations* [64]. A transaction included in the blockchain in a block with zero children has one *confirmation*. A transaction included in the blockchain in a block with N successors on the longest chain has N+1 confirmations.

Let us consider two groups of mining nodes where each group has exactly 50% of total hashing power, the group A and the group B. From long term perspective, 50% of blocks would be mined by group A and 50% of blocks would be mined by group B.

However, if group A has more than 50% of hashing power, the group would produce more blocks than group B eventually.

Majority attack

Consider a malicious actor who controls group of miners with more than 50% of total hashing power. Let be X a block know to all nodes in the network. The controlled group starts to mint secret chain from block X – the secret chain is not being published to other miners.

Let us suppose that mining difficulty is not changed from block X. The secret chain will be eventually the longest chain and the chain with the greatest cumulative difficulty.

As soon as the secret chain has the greatest cumulative difficulty, it can be propagated to other miners block by block at the same time. All miners set the secret chain as new main chain, effectively replacing all blocks from block X.

This attack is known as Majority attack, >50% attack or 51% attack [65]. Principally equal attack by malicious actor with less or equal to 50% of total hashing power is possible with lower probability of success. The generalized attack is known as Brute force attack [65]. With Majority attack, attacker can [66]:

- Make any transaction included after block X unconfirmed
- Make any transaction included in or before block X unconfirmed
- Reverse own transaction – replace own transaction with double-spent one included in secret chain
- Prevent any transaction to be permanently included in blockchain

However, with Majority attack, attacker can not [66]:

- Reverse transaction an attacker can not spend
- Censor unconfirmed transactions from network
- Steal bitcoins (spend transactions of others to attacker)
- Make other nodes to accept new consensus rules such as higher block rewards

Consider a miner with 10% of total hashing power. The miner has 20% probability to perform successful double-spend attempt on transaction with one confirmation [67]. The same miner has 5.6% probability of success with transaction with two confirmations and 0.059% probability of success with transaction with six confirmations.

However, mining of secret chain that will not get included in the blockchain costs miner operator wasted electricity. Furthermore, it costs operators block rewards and transaction fees in the non-included secret chain. Moreover, miner operators have economic incentive to not to attack Bitcoin if they invested in mining hardware and expect return of investment or future profit. Decentralized mining together with economic incentives maintain practical blockchain immutability.

From the perspective of bitcoin recipients, they need to estimate number of confirmations C such that C is high enough for them to consider received transaction to be immutable. For example, the probability of successful double-spend attack with successfully reversed transaction is less if C=6 than if C=1 [67]. The number of sufficient confirmations can be decided by a bitcoin recipient for various transaction individually to be exposed to acceptable fraud impact.

At consensus level, coinbase transactions with less than COINBASE_MATURITY (COINBASE_MATURITY = 100 as of 2016) confirmations can not be spend [68]. One reason for such a high number of needed confirmations is that coinbase transaction with created bitcoins can not be re-included into other block if main chain reconverges.

3 Contracts

Contract is a voluntary arrangement between two or more parties that is enforceable at law as a binding legal agreement [69]. Contracts usually identify all parties and contain semantics expressed in human language. Trusted party can sign the contract to protect contract authenticity and authorities can forcefully implement actions defined in a contract.

When digitized, contract parties can be represented as identity identifiers, contract semantics in human language can be stored in computer code as conditions, rules and procedures. To avoid confusion with terminology of digitalized contracts, we present two different [70] yet related contract concepts as they were originally defined.

3.1 Ricardian contract

Ricardian contract is a document format for contracts readable by humans and computers [71] as well as Ricardian contract is a software design pattern [72]. It was invented by Ian Grigg in 1995 – 1996 to carry issued financial instruments over the Internet [73, 74]:

> A Ricardian Contract can be defined as a single document that is a) a contract offered by an issuer to holders, b) for a valuable right held by holders, and managed by the issuer, c) easily readable by people (like a contract on paper), d) readable by programs (parsable like a database), e) digitally signed, f) carries the keys and server information, and g) allied with a unique and secure identifier. [74]

3.2 Smart contract

Smart contract is a protocol that facilitate, verify, or enforce the negotiation of a contract [75]. The term was defined and multiple applications were described by Nick Szabo in 1994 – 1997 [7, 76]:

> A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives

> of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs. [7]

3.3 Contract examples

In this section we describe (human) contracts in more detail. Generally, contracts require a good will from each involved party to accept the contract as valid. Similarly, a good will is required to implement actions defined by the contract. Accepted authorities can forcefully substitute the good will. However, involvement of such authorities can have high implementation cost and authorities can be dishonest. A contract is enforced by the norms and the law from the Pathetic dot theory [77].

Contracts can be formalized and digitalized into computer code in a programming language and stored on a personal computer, a server or a cloud platform. The code can be used to evaluate input information as contract rules are defined and output exact answers. A smart contract is enforced by the architecture and the market from the Pathetic dot theory [77].

Bitcoin can be used as programming platform to store and execute formalized contracts. However, in contrast with other centralized systems and platforms, Bitcoin platform is not controlled by a single authority or chain of authorities.

3.3.1 Frozen deposit

Consider an user and a service. User wants to register account to the service and the service requires proof that the user is not bot. However, the service does not want to sell registrations. The following contract can be created:

- User allocates and freezes C coins.
- C coins which will be frozen for D days.

- Frozen C coins can not be spend.
- After D days, C coins will be unfrozen automatically.
- Frozen C coins can be unfrozen at any time if both the user and the service agree to.

The service can never spend frozen coins, however it received some guarantees that the user is not a bot (we suppose that bots do not pay B coins).

After D days, the user can be automatically removed from the service, or contract can be modified to allow to freeze allocated coins again. Alternatively, the user could be considered as human by behavioral analysis and allowed to be registered without frozen coins.

If user wants to cancel own account, both parties can agree to unfrozen coins. If the service does not respond to cancel request, user will receive all coins later on day D.

Conceptually equal contract can be implemented in Bitcoin as Smart contract. However, the implementation and realization are not straightforward – it would require to use transaction sequence number, it would require to include exchange of public keys between the two participants and it would require to create two chained transactions [78].

3.3.2 Escrow and dispute mediation

Consider a buyer, a seller and a mediator. For example, seller sells hardware wallets. However, the buyer lives 500 km away from the seller and both have low reputation. The mediator has high reputation.

- Buyer allocates C coins.
- Allocated C coins can not be spend.
- If buyer and seller agree, C coins are accounted to seller.
- If buyer and mediator agree, C coins are accounted to buyer.
- If seller and mediator agree, C coins are accounted to seller.
- In other cases, C coins are still allocated.

- As soon as C coins are accounted, reputation of mediator is increased.

If reputation is valuable enough, the mediator has incentive to resolve potential dispute eventually.

The contract can be further refined. For example, some coins can be accounted to the mediator as soon as dispute is resolved. This would increase short-term incentive for mediator. Furthermore, is dispute occurs, allocated coins could be split between buyer and seller after D days automatically.

The contract can be implemented in Bitcoin as Smart contract with Multi-Signature authentication scheme 2-of-3 [79].

4 Ethereum

Ethereum is a decentralized platform that runs smart contracts. In this chapter we describe Ethereum Homestead milestone release.

Similarly to Bitcoin, Ethereum uses internal tokens called ethers with atomic unit wei (1 ether = 10^{18} wei). The issuance model of ethers is disinflationary [2] with maximum value (not constant) for a block.

As of 2016, Etherem blockchain is being constructed by mining nodes using proof-of-work algorithm Ethash. Average time to find a block is 15 seconds. Block reward is 5 ethers (constant) for creator of candidate block and 5*7/8 ethers for creator of uncle block (valid but late block which is at most 6 blocks back). Miners can process and include at most two uncle blocks into own candidate block. For doing so, they get extra 1/32 ethers for each uncle block. Finally, winning miner collects transaction fees [80].

Proof-of-stake (PoS) scheme is planned to replace PoW mining scheme in future milestone release [2]. PoS scheme uses number of temporarily locked tokens (stake) as node weight and weighted pseudorandom function to select winning node.

4.1 Concepts

Although both Bitcoin and Ethereum are peer-to-peer-networks and use main concepts such as mining and distributed blockchain, internal concepts and implementation differs between both systems. However, these differences make Ethereum more suitable to program smart contract with complex logic.

4.1.1 Accounts

Bitcoin uses paradigm of transaction outputs to represent state of value belongings. Ethereum uses an alternative paradigm of *accounts*. Furthermore, Bitcoin uses unlocking scripts to define advanced authentication schemes to spend bitcoin tokens. Ethereum uses programs called contracts to define advanced authentication schemes and other programmable logic. There are two abstract types of accounts.

EOA
Address
Balance
Nonce

Table 4.1: Externally owned account [81, 2]

Externally owned account

Account of this type (table 4.1) is controlled by private key and identified by address derived from corresponding public key with cryptographic hash function SHA3 [82]. To change balance of an account, cryptographic signature with private key corresponding to public key must be provided, similarly to Bitcoin Pay-to-Public-Key-Hash. Externally owned accounts do not support advanced authentication schemes [81].

Contract account
Address
Balance
Contract code
Contract storage
Nonce

Table 4.2: Contract account [83, 2]

Contract accounts

Account of this type (table 4.2) is identified by address. However, the address is not derived from known public key. Rather it is deterministically derived from an address of known externally owned account. Cryptographic function SHA3 is used in this process [84]. As a consequence, balance of contract account can not be controlled by a corresponding private key. Instead, contract account contains computer program [83, 85] and persistent data storage which together define account behaviour.

4.1.2 Transactions

Similarly to Bitcoin, Ethereum transaction [86] can send value from one externally owned account to another externally owned account [87]. As table 4.3 shows, transaction represents change of state to an account identified by address. Transaction is valid if the sender account has not less balance than $value + fee$ and if the transaction contains transaction signature made with private key of the sender externally owned account. We describe transaction fees in more detail further in this chapter.

Transaction
recipient address value data start gas gas price
transaction signature

Table 4.3: Transaction structure [87, 2]

Differently to Bitcoin, Ethereum transaction from externally contract account can create new contracts accounts. New contract accounts are stored in Ethereum blockchain.

Finally, transaction can be send from externally owned account to existing foreign contract account. Transaction structure contains virtual transaction called *message* which is stored in *data* field of transaction as table 4.3 shows.

4.1.3 Messages

Messages (appendix A in [88]) are data objects exchanged between accounts of both types. Although messages are structurally similar to transactions as table 4.4 shows, they are not stored in the blockchain as transactions are.

Message sent from externally owned account to contract account represents transfer of value. Moreover, in this case the message invokes contract code contained in the destination contract account. Message is analogically similar to program procedure or function.

Message
sender address
recipient address
value
data
start gas

Table 4.4: Message structure [87, 2]

Similarly, message can be send from contract account to another contract account. In this case, the analogy of program procedure or function is even clearer.

4.1.4 Smart contracts

Ethereum contract or Smart contract in Ethereum is collection of computer code and data included in contract account on Ethereum blockchain [83]. Although contracts can be used to implement authentication schemes for spending account balance similarly to Bitcoin, there are significant differences between Bitcoin script programs and Ethereum contracts [2].

- Contracts are quasi-Turing-complete (chapter 9 in [88])
- Contracts are not stateless, they can persist computed data
- Contracts can access some data of other contracts

4.1.5 Ethereum Virtual Machine (EVM)

Ethereum Virtual Machine is the runtime environment for Ethereum contracts. EVM software is included on network nodes which execute contract bytecode. The execution is isolated from non-idempotent sources such as network and file system [89].

Each Ethereum block stores a state root of Patricia tree [90]. State root is hash comprising state of every account and every contract storage ever created and included in block [91, 92].

When a miner finds candidate block with included transactions, new state root is computed by processing all transactions in the candidate block. This process executes included contracts and it can invoke code of other contract accounts. In effect, state of some accounts can be changed. State root is recomputed. The recomputed state root is included into candidate block and propagated to other network nodes.

When full-node receives new block, it reconstructs state root locally by executing all transactions in the received block. If the locally computed state root is equal to the state root included in the received block then the received block is valid [93] and mining process restarts on the new block.

Although execution of all contracts is performed by each full-node independently and locally, block reward and fees are accounted only to a miner who successfully propagated own minted candidate block into the global blockchain.

Gas

In Bitcoin, miners usually prioritize transactions with higher transaction fee to maximize profit. This economic incentive protects blockchain from empty blocks. Secondly, when block reward decreases in the future, transaction fees will function as more significant incentive replacing block reward to secure Bitcoin consensus.

Ethereum implements miner fee for the same reasons as Bitcoin. However, there exists additional reason to pay miner fees. The EVM allows to run contracts with potentially infinite loops [31] or computationally expensive operations. To prevent DDoS attacks and compensate miners for time-consuming computations, contracts consume a *gas*. Gas is internal unit used in the EVM. Each EVM instruction (operation code) has assigned *gas cost*. Gas cost specifies how much gas units an operation code consumes on one execution.

Upon program execution, the EVM increments *used gas* by gas cost of each individual instruction. If used gas would exceed a *start gas* contained in transaction or message, the execution is halted.

Creator of new transaction must estimate and include into the transaction enough start gas needed to run contract code at destination account. Moreover, creator must include *gas price* in new transaction [87]. The gas price is a bid price in ETH for one gas unit. Miners usually prioritize transactions with higher gas price. When execution terminates properly, total cost of $used\_gas \times gas\_price$ ETH is subtracted from source account balance. When there is not enough gas for next

Instruction name	Gas cost	Description
STOP	0	halt execution
SUICIDE	0	as STOP and register account for deletion
SHA3	20	compute Keccak-256 hash
SLOAD	20	load word from permanent storage
SSTORE	100	save word to permanent storage
BALANCE	20	get balance of the given account
CREATE	100	create a new account with associated code
CALL	20	message-call into an account
Process name		
step	1	one execution cycle
memory	1	additional word when expanding memory
txdata	5	byte of data or code for a transaction
contract	53000	create contract from transaction

Table 4.5: Subset of instructions, processes and gas cost (appendix H in [88] and [87])

instruction, the total cost is subtracted from source account balance and other changes made during the execution are reverted.

High-level programming languages

Programs for the Ethereum Virtual Machine are usually written in high-level programming language such as Solidity [94] or Serpent [95]. In this chapter we use Solidity language for code examples.

In this chapter we present realistic problem and propose solution to the problem. Later we present conclusion

4.2 Household problem

In this section we present simplified yet realistic problem as well as solution to the problem implemented as Ethereum smart contract.

Consider a person who owns an apartment. Another N people rent one room in the apartment, each person rents one room. Each resident is responsible for cleanliness of own room. However, there are D commonly used rooms used by all residents, such as entrance hall, kitchen, living room and bathroom. Each room must be in clean state for R days. Generally, there are N people and D duties for each round of R days.

The owner do not want to solve complaints and conflicts between residents regarding the quality of cleanliness. Moreover the owner do not want to watch over duty assignments.

Instead, the owner can formulate a contract, formalize the contract as smart contract in Solidity language deploy it to the Ethereum. The owner is willing to periodically send one message to the smart contract as new round reminder. If special event occurs, such as new resident must be registered, the owner is willing to take more actions.

4.2.1 The contract

When new person is accommodated, it deposits B value to apartment owner. The owner can not steal the deposit. However, the deposit deposit can be locked and unlocked by the owner. Residents can access only unlocked deposit.

Assignment of duties is changed on each round – each person will have different duty next round. Circular assignment with static resident and duty orders will be used. New people can be accommodated and un-accommodated in the future or duties can be added or removed. If necessary, the owner can reset assignments to default state.

If some resident is not satisfied with cleanliness of commonly used room, he or she can complain once for a round. One complaint cost specified value. The value can be changed by the owner for next round.

When current round finishes, all duty assignments are checked. If there is an assignment on which at least 50% of residents had complained then the accumulated cost of all assignment complaints is subtracted from the deposit balance of resident who is responsible for the duty. If there are some complaints in not sufficient count (less than 50%) then the accumulated balance is accounted to the private account of the owner.

4.2.2 The smart contract

The smart contract we constructed in Solidity language is pseudo-equivalent of the contract above. The source code is included as appendix A. The smart contract was deployed from two accounts:

- Main network: E2684b3e13504cb6bb008aA2342897040fBdCd37
- Test network: d574aeb383e18baca86a82b6aa28555f0a394277

New contract accounts were automatically created:

- Main network: A7a4db3AC7822f988d50827f975545AF4E97954F
- Test network: 514075560BD774460615b59124BC02390273ACd3

All four accounts are stored in Ethereum blockchain. Web service etherchain.org can be used to search for addresses on the main network and blockchain. Similarly, testnet.etherscan.io can search addresses on the test network and blockchain.

Both externally owned accounts can be imported into wallet software. See appendix B for corresponding private keys and further instructions. For list of development tools see appendix C.

4.2.3 Feasibility and clarity

In software engineering it is difficult to precisely specify functional requirements and create equivalent software. In fact, a contract is set of functional requirements and smart contract is the corresponding implementation.

An ambiguous contract can be interpreted differently by one or more people. An absurd interpretation or requirement can be intuitively omitted, preferred interpretation can be selected by authority or by voting process. For example, the Household contract contains the following text:

> However, the deposit deposit can be locked and unlocked
> by the owner. Residents can access only unlocked deposit.

There are at least two interpretations. The first one is that each resident can access each unlocked deposit of each resident. The second interpretation is that each resident can access only own deposit if the deposit is unlocked. However, the first one is absurd and would be intuitively omitted.

If absurd situation is defined as part of a smart contract or absurd situation occurs as a contract state, the smart contract defines the exact meaning and what actions of repair can be done.

4.3 Hack of The DAO

Decentralized autonomous organization is organization run through rules defined by smart contract [96]. The implemented Household contract can be considered as small DAO.

One example of popular DAO is The DAO, a form of investor-directed venture capital fund which raised 10.7 millions ether in market value of $120 at the time [97].

In June 2016, security vulnerability in contract logic of The DAO was exploited [98]. We describe simplified version of the vulnerability.

A function in contract account sends back ethers to contract account of another user and then updates balance counter of the user. However, the attacker created contract account which calls back the vulnerable function when ethers are received, effectively creating mutual recursion in one program instance. The fact that balance counter was decreased after the recursion call allowed to withdraw more ethers than value of attacker's balance counter.

However, the vulnerability of balance update and recursive calls were one part of more complex logic. In reality, drained ethers were allocated to temporarily locked for 28 days in child DAO.

An attacker drained 3.6 million ether (around $70 million at the time) [99] into child DAO where they had to be for 28 days. Ethereum developers proposed and implemented controversial patch [100] in Ethereum software which replaced attacker's transaction with refund transaction to safe contract account. However, such patch knowingly split Ethereum blockchain into two chains (hard-fork) with two separate currencies.

The original Ethereum chain is being constructed and maintained by nodes using Ethereum Classic software. Developers of Ethereum classic created a manifesto declaring that immutability and neutrality are necessary [101].

As of December 2016, both Ethereum (ETH) and Ethereum Classic (ETC) have relatively high market capitalization and belong to the market capitalization chart of top 10 crypto-currencies [3].

5 Conclusion

We can conclude that Solidity language, Ethereum Virtual Machine as well as developer tools such as Ethereum Studio offer convenient contract development in contrast with Bitcoin contract development.

However, during the Household contract implementation we found that a relatively simple contract code can contain security vulnerabilities such as invoking an unknown (potentially recursive) contract by sending ethers to the contract if changes to the global contract state are made after the send function.

Correctness is a critical property of smart contracts – incorrectly implemented program due to ambiguous program specification or software bugs can result in permanent lock, or undesired accessibility of ether tokens. Best practices should be learned by developers and new software design patterns should be proposed for smart contract development.

The vulnerability in The DAO contract and its exploitation which resulted in initial loss of $70 million and the instigation of the controversial deliberate hard-fork, support our conclusion. However, both Ethereum and Ethereum Classic cryptocurrencies are actively traded as of December 2016. More importantly, such stress events have beneficial effect on antifragility of decentralized systems.

Bibliography

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: `https://bitcoin.org/bitcoin.pdf` (Retrieved 03/12/2016).

[2] Vitalik Buterin. *White Paper*. URL: `https://github.com/ethereum/wiki/wiki/White-Paper` (Retrieved 12/05/2016).

[3] CoinMarketCap. *Crypto-Currency Market Capitalizations*. URL: `https://coinmarketcap.com/` (Retrieved 12/29/2016).

[4] CoinDesk. *What is the Difference Between Litecoin and Bitcoin?* URL: `http://www.coindesk.com/information/comparing-litecoin-bitcoin/` (Retrieved 12/30/2016).

[5] *What is Zcash?* URL: `https://github.com/zcash/zcash/blob/master/README.md` (Retrieved 12/30/2016).

[6] Wikipedia. *CryptoNote*. URL: `https://en.wikipedia.org/wiki/CryptoNote#Current_CryptoNote_currencies` (Retrieved 12/30/2016).

[7] Nick Szabo. *Smart Contracts*. URL: `http://archive.is/X3lR2` (Retrieved 12/22/2016).

[8] Eurostat Press Office. *E-commerce by individuals*. URL: `http://ec.europa.eu/eurostat/documents/2995521/7103356/4-11122015-AP-EN.pdf` (Retrieved 03/12/2016).

[9] Swedish Institute. *Sweden – the first cashless society?* URL: `https://sweden.se/business/cashless-society/` (Retrieved 05/04/2016).

[10] Ingrid Melander. *France steps up monitoring of cash payments to fight 'low-cost terrorism'*. URL: `http://www.reuters.com/article/us-france-security-financing-idUSKBN0ME14720150318` (Retrieved 03/18/2016).

[11] Todd White. *Spain Cash-Transaction Ban Begins as Rajoy Targets Tax Fraud*. URL: `http://www.bloomberg.com/news/articles/2012-11-19/spain-cash-transaction-ban-begins-as-rajoy-targets-tax-fraud` (Retrieved 03/14/2016).

[12] Philip Oltermann. *German plan to impose limit on cash transactions met with fierce resistance*. URL: `http://www.theguardian.com/world/2016/feb/08/german-plan-prohibit-large-5000-cash-transactions-fierce-resistance` (Retrieved 03/13/2016).

[13] Wikipedia. *Society for Worldwide Interbank Financial Telecommunication*. URL: https://en.wikipedia.org/wiki/Society_for_Worldwide_Interbank_Financial_Telecommunication (Retrieved 03/18/2016).

[14] Hui chin Chen. *International Transactions 101 – Credit Card*. URL: http://moneymattersforglobetrotters.com/international-transactions-101-credit-card/ (Retrieved 03/18/2016).

[15] Wikileaks. *Banking Blockade*. URL: https://wikileaks.org/Banking-Blockade.html (Retrieved 03/18/2016).

[16] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, Inc., 2014. ISBN: 978-1-449-37404-4.

[17] Bitcoin Project. *Choose your Bitcoin wallet*. URL: https://bitcoin.org/en/choose-your-wallet (Retrieved 03/12/2016).

[18] *ASIC and FPGA miner in c for bitcoin*. URL: https://github.com/ckolivas/cgminer (Retrieved 09/20/2016).

[19] Bitcoin Wiki. *BFGMiner*. URL: https://en.bitcoin.it/wiki/BFGMiner (Retrieved 09/20/2016).

[20] Bitcoin Project. *Running A Full Node*. URL: https://bitcoin.org/en/full-node (Retrieved 09/22/2016).

[21] Bitcoin Project. *Full Validation*. URL: https://bitcoin.org/en/bitcoin-core/features/validation (Retrieved 09/12/2016).

[22] blockchain.info. *Blockchain Size*. URL: https://blockchain.info/charts/blocks-size?timespan=all (Retrieved 09/20/2016).

[23] J. Postel. *Computer Mail Meeting Notes*. URL: https://tools.ietf.org/html/rfc805 (Retrieved 03/12/2016).

[24] Bitcoin Wiki. *Address*. URL: https://en.bitcoin.it/wiki/Address (Retrieved 05/02/2016).

[25] Antoon Bosselaers. *The hash function RIPEMD-160*. URL: https://homes.esat.kuleuven.be/~bosselae/ripemd160.html (Retrieved 05/11/2016).

[26] National Institute of Standards Information Technology Laboratory and Technology. *Secure Hash Standard (SHS)*. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (Retrieved 05/11/2016).

[27] The Bitcoin Core developers. *EncodeBase58Check*. URL: https://github.com/bitcoin/bitcoin/blob/57704499be948c640c789c7fc11ed1abf8a681bd/src/base58.cpp#L117 (Retrieved 05/11/2016).

[28] Bitcoin Wiki. *Protocol rules*. URL: https://en.bitcoin.it/wiki/Protocol_rules (Retrieved 12/22/2016).

[29] Cryptocompare. *How does a Bitcoin node verify a transaction?* URL: https://www.cryptocompare.com/coins/guides/how-does-a-bitcoin-node-verify-a-transaction/ (Retrieved 12/22/2016).

[30] Bitcoin Wiki. *Script*. URL: https://en.bitcoin.it/wiki/Script (Retrieved 11/17/2016).

[31] Wikipedia. *Halting problem*. URL: https://en.wikipedia.org/wiki/Halting_problem (Retrieved 12/05/2016).

[32] Bitcoin Project. *Pubkey Script, ScriptPubKey*. URL: https://bitcoin.org/en/glossary/pubkey-script (Retrieved 11/08/2016).

[33] Bitcoin Project. *Signature Script, ScriptSig*. URL: https://bitcoin.org/en/glossary/signature-script (Retrieved 11/08/2016).

[34] Bitcoin Wiki. *Script - Obsolete pay-to-pubkey transaction*. URL: https://en.bitcoin.it/wiki/Script#Obsolete_pay-to-pubkey_transaction (Retrieved 06/09/2016).

[35] Bitcoin Project. *Standard Transaction*. URL: https://bitcoin.org/en/glossary/standard-transaction (Retrieved 09/23/2016).

[36] Bitcoin Project. *Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#standard-transactions (Retrieved 11/05/2016).

[37] Gavin Andresen. *Proposal: open up IsStandard for P2SH transactions*. URL: https://gist.github.com/gavinandresen/88be40c141bc67acb247 (Retrieved 09/22/2016).

[38] Bitcoin Wiki. *OP_CHECKSIG*. URL: https://en.bitcoin.it/wiki/OP_CHECKSIG (Retrieved 06/09/2016).

[39] Christof Zalka John Proos. *Shor's discrete logarithm quantum algorithm for elliptic curves*. URL: https://arxiv.org/abs/quant-ph/0301141 (Retrieved 11/03/2016).

[40] Vitalik Buterin. *Bitcoin Is Not Quantum-Safe, And How We Can Fix It When Needed*. URL: https://bitcoinmagazine.com/articles/bitcoin-is-not-quantum-safe-and-how-we-can-fix-1375242150 (Retrieved 11/02/2016).

[41] Gavin Andresen. *Pay to Script Hash*. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (Retrieved 09/21/2016).

[42] Soroush Pour. *Bitcoin multisig the hard way: Understanding raw P2SH multisig transactions*. URL: `http://www.soroushjp.com/2014/12/20/bitcoin-multisig-the-hard-way-understanding-raw-multisignature-bitcoin-transactions/` (Retrieved 09/12/2016).

[43] almel. *Explanation of what an OP_RETURN transaction looks like*. URL: `http://bitcoin.stackexchange.com/a/29555` (Retrieved 12/15/2016).

[44] Bitcoin Wiki. *Block hashing algorithm*. URL: `https://en.bitcoin.it/wiki/Block_hashing_algorithm` (Retrieved 05/06/2016).

[45] Blockchain.info. *Block #0*. URL: `https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f` (Retrieved 12/02/2016).

[46] Bitcoin Project. *Merkle Tree*. URL: `https://bitcoin.org/en/glossary/merkle-tree` (Retrieved 09/19/2016).

[47] blockchain.info. *Block Height 0*. URL: `https://blockchain.info/block-height/0` (Retrieved 09/20/2016).

[48] blockchain.info. *Block Height 1*. URL: `https://blockchain.info/block-height/1` (Retrieved 09/20/2016).

[49] blockchain.info. *Block Height 2*. URL: `https://blockchain.info/block-height/2` (Retrieved 09/20/2016).

[50] Bitcoin Project. *Consensus rules*. URL: `https://bitcoin.org/en/glossary/consensus-rules` (Retrieved 09/29/2016).

[51] Bitcoin Project. *Consensus*. URL: `https://bitcoin.org/en/glossary/consensus` (Retrieved 09/29/2016).

[52] Bitcoin Project. *Hard fork*. URL: `https://bitcoin.org/en/glossary/hard-fork` (Retrieved 09/20/2016).

[53] Bitcoin Wiki. *Protocol rules*. URL: `https://en.bitcoin.it/wiki/Protocol_rules#.22block.22_messages` (Retrieved 12/23/2016).

[54] Bitcoin Project. *Transaction Fees And Change*. URL: `https://bitcoin.org/en/developer-guide#term-high-priority-transactions` (Retrieved 09/25/2016).

[55] Bitcoin Wiki. *Block size limit controversy*. URL: `https://en.bitcoin.it/wiki/Block_size_limit_controversy` (Retrieved 12/22/2016).

[56] Bitcoin Project. *Minimum Relay Fee*. URL: `https://bitcoin.org/en/glossary/minimum-relay-fee` (Retrieved 09/26/2016).

[57] Bitcoin Foundation. *Floating Fees for 0.10*. URL: `http://bitcoinfoundation.org/floating-fees-for-0-10/` (Retrieved 09/26/2016).

[58] Bitcoin Wiki. *Controlled supply*. URL: `https://en.bitcoin.it/wiki/Controlled_supply` (Retrieved 09/25/2016).

[59] Bitcoin Wiki. *Proof of work*. URL: `https://en.bitcoin.it/wiki/Proof_of_work` (Retrieved 12/23/2016).

[60] Bitcoin Wiki. *Hashcash*. URL: `https://en.bitcoin.it/wiki/Hashcash` (Retrieved 09/25/2016).

[61] Wikipedia. *Cryptographic hash function*. URL: `https://en.wikipedia.org/wiki/Cryptographic_hash_function` (Retrieved 05/02/2016).

[62] Wikipedia. *Collision resistance*. URL: `https://en.wikipedia.org/wiki/Collision_resistance` (Retrieved 05/02/2016).

[63] Satoshi Nakamoto. *Re: Bitcoin P2P e-cash paper*. URL: `http://satoshi.nakamotoinstitute.org/emails/cryptography/5/#selection-33.55-37.15` (Retrieved 12/26/2016).

[64] Bitcoin Project. *Confirmation Score, Confirmed Transaction*. URL: `https://bitcoin.org/en/glossary/confirmation-score` (Retrieved 11/05/2016).

[65] Bitcoin Wiki. *Double-spending*. URL: `https://en.bitcoin.it/wiki/Double-spending` (Retrieved 12/23/2016).

[66] Bitcoin Wiki. *Weaknesses*. URL: `https://en.bitcoin.it/wiki/Weaknesses#Attacker_has_a_lot_of_computing_power` (Retrieved 12/23/2016).

[67] Meni Rosenfeld. *Analysis of hashrate-based double-spending*. URL: `https://bitcoil.co.il/Doublespend.pdf` (Retrieved 12/22/2016).

[68] Bitcoin Wiki. *Protocol rules*. URL: `https://en.bitcoin.it/wiki/Protocol_rules#.22tx.22_messages` (Retrieved 11/05/2016).

[69] Wikipedia. *Contract*. URL: `https://en.wikipedia.org/wiki/Contract` (Retrieved 12/22/2016).

[70] Ian Grigg. *On the intersection of Ricardian and Smart Contracts*. URL: `http://iang.org/papers/intersection_ricardian_smart.html` (Retrieved 12/22/2016).

[71] Ian Grigg. *The Ricardian Contract*. URL: `http://iang.org/ricardian/` (Retrieved 12/22/2016).

[72] webfunds.org. *Ricardian contracts*. URL: `http://webfunds.org/guide/ricardian.html` (Retrieved 12/22/2016).

[73] iang. *IP concerns over Ricardian contracts*. URL: `http://financialcryptography.com/mt/archives/001595.html` (Retrieved 12/22/2016).

[74] Ian Grigg. *The Ricardian Contract*. URL: http://iang.org/papers/ricardian_contract.html (Retrieved 12/22/2016).

[75] Wikipedia. *Smart contract*. URL: https://en.wikipedia.org/wiki/Smart_contract (Retrieved 12/22/2016).

[76] Nick Szabo. *The Idea of Smart Contracts*. URL: http://archive.is/20DNf (Retrieved 12/22/2016).

[77] Wikipedia. *Pathetic dot theory*. URL: https://en.wikipedia.org/wiki/Pathetic_dot_theory (Retrieved 12/20/2016).

[78] Bitcoin Wiki. *Contract - Providing a deposit*. URL: https://en.bitcoin.it/wiki/Contract#Example_1:_Providing_a_deposit (Retrieved 12/20/2016).

[79] Bitcoin Wiki. *Contract - Escrow and dispute mediation*. URL: https://en.bitcoin.it/wiki/Contract#Example_2:_Escrow_and_dispute_mediation (Retrieved 12/20/2016).

[80] Joseph Lubin. *The Issuance Model in Ethereum*. URL: https://blog.ethereum.org/2014/04/10/the-issuance-model-in-ethereum/ (Retrieved 12/20/2016).

[81] Ethereum community. *Account Management*. URL: http://ethdocs.org/en/latest/account-management.html (Retrieved 12/07/2016).

[82] tayvano. *How are ethereum addresses generated?* URL: http://ethereum.stackexchange.com/a/3619 (Retrieved 12/08/2016).

[83] Ethereum community. *Contracts*. URL: http://ethdocs.org/en/latest/contracts-and-transactions/contracts.html (Retrieved 12/07/2016).

[84] eth. *How is the address of an Ethereum contract computed?* URL: http://ethereum.stackexchange.com/a/761 (Retrieved 12/08/2016).

[85] Jeff Coleman. *Where is the private key for a contract stored?* URL: http://ethereum.stackexchange.com/a/193 (Retrieved 12/08/2016).

[86] jimkberry. *What is the ethereum transaction data structure?* URL: http://ethereum.stackexchange.com/a/2097 (Retrieved 12/08/2016).

[87] Ethereum community. *Account Types, Gas, and Transactions*. URL: http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html (Retrieved 12/08/2016).

[88] Dr. Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. URL: `http://paper.gavwood.com` (Retrieved 12/15/2016).

[89] Ethereum community. *The EVM*. URL: `http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#the-evm` (Retrieved 12/09/2016).

[90] stephantual. *Patricia Tree*. URL: `https://github.com/ethereum/wiki/wiki/Patricia-Tree` (Retrieved 12/20/2016).

[91] Vitalik Buterin. *State Tree Pruning*. URL: `https://blog.ethereum.org/2015/06/26/state-tree-pruning/` (Retrieved 12/15/2016).

[92] CJentzsch $\mathrm{x}_E THeREAL_x work2heat$. *Basic Questions About the Ethereum EVM and State Storage*. URL: `https://www.reddit.com/r/ethereum/comments/3k4h3w/basic_questions_about_the_ethereum_evm_and_state/` (Retrieved 12/15/2016).

[93] Smithgift sherry_nicta. *how is contract validated*. URL: `https://forum.ethereum.org/discussion/2324/how-is-contract-validated` (Retrieved 12/15/2016).

[94] Ethereum. *Solidity in Depth*. URL: `http://solidity.readthedocs.io/en/latest/solidity-in-depth.html` (Retrieved 12/26/2016).

[95] Joseph Krug. *Serpent*. URL: `https://github.com/ethereum/wiki/wiki/Serpent` (Retrieved 12/26/2016).

[96] Ethereum Foundation. *Decentralized Autonomous Organization*. URL: `https://www.ethereum.org/dao` (Retrieved 12/29/2016).

[97] Richard Waters. *Automated company raises equivalent of $120M in digital currency*. URL: `http://www.cnbc.com/2016/05/17/automated-company-raises-equivalent-of-120-million-in-digital-currency.html` (Retrieved 12/29/2016).

[98] Phil Daian. *Analysis of the DAO exploit*. URL: `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/` (Retrieved 12/29/2016).

[99] Cryptocompare. *The DAO, The Hack, The Soft Fork and The Hard Fork*. URL: `https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/` (Retrieved 12/29/2016).

[100] Jeffrey Wilcke. *To fork or not to fork*. URL: `https://blog.ethereum.org/2016/07/15/to-fork-or-not-to-fork/` (Retrieved 12/29/2016).

[101] Arvicco. *A Crypto-Decentralist Manifesto*. URL: `https://ethereumclassic.github.io/blog/2016-07-11-manifesto/` (Retrieved 12/29/2016).

A Household smart contract

```
pragma solidity ^0.4.0;

contract Household {

  /*
  * ************************************************
  * Structures
  * ************************************************
  */

  struct Resident {
    uint balance; // deposit
    address aPrev; // linked list pointer
    address aNext; // linked list pointer
    bool exists; // never set to false in code
    bool accomodated; // lives in household
    bool locked; // can not withdraw own balance
  }

  struct Complaint {
    uint balance; // counter of total complaints
    uint8 count; // number of complaints
  }

  struct RoundParameters {
    uint complaintCost; // cost of one complaint in a round
    uint8 dutyCount; // number of duties in a round
  }

  struct Round {
    bool needsDutyReset; // inconsistency of next round
    bool running; // round in progress
    RoundParameters parameters;
    mapping(address => uint8) assignments; // resident => has duty
    mapping(uint8 => Complaint) complaints; // duty => has complaint
    mapping(address => bool[]) complained; // resident => complained
  } Round round;

  /* Household owner and contract owner */
  struct Owner {
    address addr; // external address
    uint balance; // counter of balance
  } Owner owner;

  struct Parameters {
    uint minDepositBalance; // threshold
    RoundParameters round;
  } Parameters parameters;

  /* Linked list of residents */
  struct Residents {
    address first; // first resident address
    address last; // last resident address
    uint accomodatedCount; // number of accomodated residents
    uint existsCount; // number of past residents
    mapping(address => Resident) map; // residents
  } Residents residents;

  /*
  * *************************************************
  * Modifiers
  * *************************************************
  */

  modifier onlyOwner() {
    if (msg.sender != owner.addr) throw;
    _;
  }

  /*
  * *************************************************
  * Public functions
  * *************************************************
  */

  /* Constructor */
  function Household(uint minDepositBalance, uint complaintCost,
    uint8 dutyCount)
  {
    owner.addr = msg.sender;
    setMinDepositBalance(minDepositBalance);
    setComplaintCost(complaintCost);
    setDutyCount(dutyCount);
  }

  /* Accomodates new resident */
```

```
function accomodate(address raddr) onlyOwner {
  if (residents.accomodatedCount + 1 >
    parameters.round.dutyCount) throw;

  if (!exists(raddr)) {
    create(raddr);
  }
  residents.map[raddr].accomodated = true;
  residents.accomodatedCount++;
  residents.map[raddr].locked = true;
}

/* Unaccomodates a resident */
function unaccomodate(address raddr) onlyOwner {
  if (!exists(raddr)) throw;
  if (!accomodated(raddr)) return;
  Resident r = residents.map[raddr];
  if (raddr == residents.first && raddr == residents.last) {
    residents.first = 0;
    residents.last = 0;
  } else {
    if (raddr == residents.first) {
      residents.map[residents.map[raddr].aNext].aPrev = 0;
      residents.first = residents.map[raddr].aNext;
    } else if (raddr == residents.last) {
      residents.map[residents.map[raddr].aPrev].aNext = 0;
      residents.last = residents.map[raddr].aPrev;
    } else {
      residents.map[residents.map[raddr].aNext].aPrev =
              residents.map[raddr].aPrev;
      residents.map[residents.map[raddr].aPrev].aNext =
              residents.map[raddr].aNext;
    }
  }
  r.aPrev = 0;
  r.aNext = 0;
  r.accomodated = false;
  residents.accomodatedCount--;
  round.needsDutyReset = true;
}

/* Deposits value to the resident  */
function deposit(address raddr) payable {
  if (!exists(raddr)) throw;
  residents.map[raddr].balance += msg.value;
  }

  /* Restarts a round */
  function newRound() onlyOwner {
    settleRound();
    initRound();
  }

  /* Complains for quality of a duty */
  function complain(uint8 dutyCode) payable {
    if (!exists(msg.sender)) throw;
    if (!accomodated(msg.sender)) throw;
    if (!running()) throw;
    if (!assigned(dutyCode)) throw;
    if (msg.value != round.parameters.complaintCost) throw;
    if (complained(msg.sender, dutyCode)) return;
    round.complaints[dutyCode - 1].balance += msg.value;
    round.complaints[dutyCode - 1].count++;
    round.complained[msg.sender][dutyCode - 1] = true;
  }

  /* Withdraws balance from the contract,
   limited by resident balance counter */
  function withdraw() returns (bool) {
    if (!exists(msg.sender)) throw;
    if (locked(msg.sender)) return false;
    uint balance = residents.map[msg.sender].balance;
    residents.map[msg.sender].balance = 0;
    if (msg.sender.send(balance)) {
      return true;
    } else {
      residents.map[msg.sender].balance = balance;
      return false;
    }
  }

  /* Withdraws balance from the contract,
   limited by owner balance counter */
  function withdrawOwner() onlyOwner returns (bool) {
    uint balance = owner.balance;
    owner.balance = 0;
    if (msg.sender.send(balance)) {
      return true;
    } else {
      owner.balance = balance;
      return false;
    }
  }

  /*
  * ************************************************
  * Setters
  * ************************************************
  */

  /* Sets min deposit balance threshold */
  function setMinDepositBalance(uint minDepositBalance)
  onlyOwner {
    parameters.minDepositBalance = minDepositBalance;
  }

  /* Adds or removes duties for next rounds */
  function setDutyCount(uint8 dutyCount) onlyOwner {
    if (residents.accomodatedCount > dutyCount) throw;
    parameters.round.dutyCount = dutyCount;
    round.needsDutyReset = true;
  }

  /* Sets complaint cost for next rounds */
  function setComplaintCost(uint complaintCost) onlyOwner {
    parameters.round.complaintCost = complaintCost;
  }

  /* Locks resident's balance for withdrawal */
  function lock(address raddr) onlyOwner {
    if (!exists(raddr)) throw;
    residents.map[raddr].locked = true;
  }

  /* Unlocks resident's balance for withdrawal */
  function unlock(address raddr) onlyOwner {
    if (!exists(raddr)) throw;
    residents.map[raddr].locked = false;
  }

  /*
  * ************************************************
  * Getters
  * ************************************************
  */
```

```
/* Returns balance of a resident */
function getBalance(address raddr) returns (uint) {
  if (!exists(raddr)) throw;
  return residents.map[raddr].balance;
}

/* Returns balance counter of a resident */
function hasEnoughBalance(address raddr) returns (bool) {
  if (!exists(raddr)) throw;
  return residents.map[raddr].balance >=
     parameters.minDepositBalance;
}

/* If duty code is valid */
function valid(uint8 dutyCode) returns (bool) {
  if (dutyCode < 1) return false;
  if (dutyCode > round.parameters.dutyCount) return false;
  return true;
}

/* If duty code is assigned to some resident */
function assigned(uint8 dutyCode) returns (bool) {
  if (!valid(dutyCode)) return false;
  address a = residents.first;
  while (a != 0) {
    if (dutyCode == round.assignments[a]) return true;
    a = residents.map[a].aNext;
  }
  return false;
}

/*
* ************************************************
* Internal functions
* ************************************************
*/

/* Evaluates a round and assigns accumulated balance */
function settleRound() internal {
  if (!running()) return;
  address a = residents.first;
  while (a != 0) {
    uint8 dutyCode = round.assignments[a];
    if (dutyCode > 0) {
        uint8 complaintCount = round.complaints[dutyCode - 1].count;
        uint complaintBalance = round.complaints[dutyCode - 1].balance;
        if (100 * complaintCount / round.parameters.dutyCount > 50) {
          if (residents.map[a].balance >= complaintBalance) {
            residents.map[a].balance -= complaintBalance;
          }
        } else {
          owner.balance += complaintBalance;
        }
      }
      a = residents.map[a].aNext;
    }
  }

  /* Starts a new round */
  function initRound() internal {
    setNewRoundParameters();
    if (round.needsDutyReset) {
      resetDuties();
    }
    resetComplaints();
    changeDuties();
    round.running = true;
  }

  /* Changes duty assignments for old and new residents */
  function changeDuties() internal {
    address a = residents.first;
    while (a != 0) {
      if (round.assignments[a] == 0) {
        round.assignments[a] = round.assignments[residents.last] + 1;
      }
      round.assignments[a]++;
      if (round.assignments[a] > round.parameters.dutyCount) {
        round.assignments[a] = 1;
      }
      a = residents.map[a].aNext;
    }
  }

  /* Resets duties assignments to default state */
  function resetDuties() internal {
    round.needsDutyReset = false;
    address a = residents.first;
    uint i = round.parameters.dutyCount + 1;
    while (a != 0) {
      round.assignments[a] = uint8(i);
      i++;
      if (i > round.parameters.dutyCount) i = 1;
      a = residents.map[a].aNext;
    }
  }

  /* Resets complaints structures and balance of a round */
  function resetComplaints() internal {
    for (uint8 i = 0; i < round.parameters.dutyCount; i++) {
      round.complaints[i].count = 0;
      round.complaints[i].balance = 0;
    }
    address a = residents.first;
    while (a != 0) {
      round.complained[a].length = 0;
      round.complained[a].length = round.parameters.dutyCount;
      a = residents.map[a].aNext;
    }
  }

  /* Sets new round parameters changed in round progress */
  function setNewRoundParameters() internal {
    round.parameters.dutyCount = parameters.round.dutyCount;
    round.parameters.complaintCost = parameters.round.complaintCost;
  }

  /* Creates new resident */
  function create(address raddr) internal {
    residents.map[raddr].exists = true;
    residents.existsCount++;
    // setup resident
    residents.map[raddr].aPrev = residents.last;
    residents.map[raddr].aNext = 0;
    // update previous last
    residents.map[residents.last].aNext = raddr;
    // update linked list
    residents.last = raddr;
    if (residents.first == 0) {
      residents.first = raddr;
    }
  }

  /* If resident exist */
  function exists(address raddr) internal returns (bool) {
    return residents.map[raddr].exists;
  }

  /* If resident is accomodated */
  function accomodated(address raddr) internal returns (bool) {
    return residents.map[raddr].accomodated;
  }

  /* If resident is locked */
  function locked(address raddr) internal returns (bool) {
    return residents.map[raddr].locked;
  }

  /* If round is in progress */
  function running() internal returns (bool) {
    return round.running;
  }

  /* If a resident already complained for a duty in one round */
  function complained(address raddr, uint8 dutyCode)
  internal returns (bool) {
    return round.complained[raddr][dutyCode - 1];
  }
}
```

B Backup of accounts (private keys)

The backup of accounts with deployed Household smart contract is in JSON format and can be imported into **geth** console application or **Ethereum Wallet** application with GUI.

The passphrase to decrypt, import or unlock the main network account is **mainmuni**. The passphrase for the test network account is **testmuni**.

For more information about geth visit geth.ethereum.org and for more information about Ethereum Wallet visit github.com/ethereum/mist/releases.

Mainnet ethers for main network can be bought on cryptocurrency exchanges. However, testnet ethers for test network can be obtained for free from Ethereum faucet websites.

B.1 Main network account

```
{
  "address": "e2684b3e13504cb6bb008aa2342897040fbdcd37",
  "crypto": {
    "cipher": "aes-128-ctr",
    "ciphertext":
"2e5321959a7a6ad650b166c4473cc45380ea42a03b2fa49785a43b473a417110",
    "cipherparams": {
      "iv": "3f56a83bd87de37ee2d7c231a67506c6"
    },
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "n": 262144,
      "p": 1,
      "r": 8,
      "salt":
"ced04c7720a667bf6b12456a0c8e5c81548964f3e10333db0b6948da2c5b4921"
    },
    "mac":
"8fae275816fa58a2a0e6ae80f6fe7788a3f4d6520320f1e642d295c306baf61c"
  },
  "id": "bf4a1c0e-b0f2-444e-b4b3-6980be6428e4",
  "version": 3
}
```

B.2 Test network account

```
{
  "address": "d574aeb383e18baca86a82b6aa28555f0a394277",
  "crypto": {
    "cipher": "aes-128-ctr",
    "ciphertext":
"55572a7f4404dcdcf150af3f3d695eda450c7eac75247ba634c00fa1eadbecbb",
    "cipherparams": {
      "iv": "dfb9d8caccff19e3ba761a1e18bdb425"
    },
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "n": 262144,
      "p": 1,
      "r": 8,
      "salt":
"07ab6c631ccefc7d7e24e0b449cee1cf65ca1a7aa240f9bb5c8b9a3481406e2d"
    },
    "mac":
"c1f4d4916b18c8cd65d75bbf6082dd4fd023f78e513c0d165e02501d20dbbe3f"
  },
  "id": "7a5b2fac-e10f-4892-a162-6c87ff3f1b57",
  "version": 3
}
```

C Development tools

Contract functions can be invoked with **geth** (console application, geth.ethereum.org) or **Ethereum Wallet** (application with graphic interface, github.com/ethereum/mist/releases). However, for smart contract development and testing, an emulator is usefull. **Ethereum Studio** is an online IDE with Ethereum emulator (live.ether.camp/eth-studio). Solidity code can be compiled and analyzed in online **Browser Solidity** IDE (ethereum.github.io/browser-solidity).