



BLOCKCHAINS

BACHELOR THESIS

Mr.
Tim Käbisch

Verification of Bitcoin in the IN3 protocol

Mittweida, 2020

Faculty of **Applied Computer Sciences & Biosciences**

BACHELOR THESIS

Verification of Bitcoin in the IN3 protocol

Author:
Tim Käbisch

Study Programme:
Applied Computer Science, IT-Security

Seminar Group:
IF17wI2-B

First Referee:
Prof. Dr. – Ing. Andreas Ittner

Second Referee:
M. Sc. Steffen Kux (Blockchains LLC)

Submission:
11/02/2020

Defense/Evaluation:
Mittweida, 2020

Bibliographic Information

Käbisch, Tim: Verification of Bitcoin in the IN3 protocol, 54 pages, 25 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Applied Computer Sciences & Biosciences

Bachelor thesis, 2020

Typeset by LaTeX

Abstract

The number of Internet of Things (IoT) devices is increasing rapidly. The Trustless Incentivized Remote Node Network, in short IN3 (Incubed), enables trustworthy and fast access to a blockchain for a large number of low-performance IoT devices. Although currently IN3 only supports the verification of Ethereum data, it is not limited to one blockchain due to modularity. This thesis describes the fundamentals, the concept and the implementation of the Bitcoin verification in IN3.

I. Contents

Contents I

List of Figures II

1 Introduction 1

2 Fundamentals 2
- 2.1 Bitcoin . . . 2
  - 2.1.1 Mining . . . 2
  - 2.1.2 Difficulty Adjustment Period . . . 3
  - 2.1.3 Finality . . . 4
- 2.2 IN3 . . . 5
  - 2.2.1 Overview . . . 5
  - 2.2.2 Architecture . . . 6
  - 2.2.3 Incentivization . . . 7
  - 2.2.4 Use Case: Smart Lock . . . 8

3 Concept 9
- 3.1 Verification . . . 9
  - 3.1.1 Design . . . 9
  - 3.1.2 BIP34 . . . 10
  - 3.1.3 Blocks After BIP34 . . . 11
  - 3.1.4 Blocks Before BIP34 . . . 11
- 3.2 Proofs . . . 13
  - 3.2.1 Target Proof . . . 13
  - 3.2.2 Finality Proof . . . 14
  - 3.2.3 Transaction Proof . . . 16
  - 3.2.4 Block Number Proof . . . 17
- 3.3 Security Calculation . . . 18
  - 3.3.1 Blocks Before BIP34 . . . 18
  - 3.3.2 Blocks After BIP34 . . . 20
- 3.4 On-chain Conviction . . . 23

4 Implementation 27
- 4.1 Architecture . . . 27
- 4.2 RPC-Methods . . . 28
  - 4.2.1 getBlock . . . 29
  - 4.2.2 getBlockHeader . . . 30
  - 4.2.3 getTransaction . . . 31
  - 4.2.4 getBlockCount/getBestBlockHash . . . 32
  - 4.2.5 getDifficulty . . . 33
  - 4.2.6 proofTarget . . . 34

4.3 Cache . . . 36
4.3.1 Server-Cache . . . 36
4.3.2 Client-Cache . . . 37
4.4 Tests . . . 38
4.4.1 Server-Tests . . . 38
4.4.2 Client-Tests . . . 39
4.5 On-chain Conviction . . . 41

5 Demonstration **43**
5.1 Setup . . . 43
5.2 Examples . . . 44
5.3 Further Demonstration . . . 48

6 Conclusion **49**

A Source Code of the On-chain Conviction **50**

Bibliography **51**

II. List of Figures

2.1 Bitcoin Mining [7] . . . 2
2.2 IN3 Architecture [13] . . . 7
3.1 IN3 Request and Response . . . 10
3.2 Distance Sweet Spot . . . 12
3.3 Successful Finality Proof [20] . . . 15
3.4 Failed Finality Proof [20] . . . 15
3.5 Merkle Tree [21] . . . 16
3.6 Security of Finality Headers . . . 22
3.7 Bitcoin Longest Chain [26] . . . 24
3.8 Game On-chain Conviction . . . 25
4.1 Architecture . . . 27
4.2 Visualization of proofTarget . . . 35
4.3 Demonstration of the Cache . . . 36
4.4 IN3 Server Code Coverage [30] . . . 39
4.5 IN3 Client Code Coverage . . . 40
4.6 Soldity Code *verifyHeader* . . . 41
5.1 Setup WASM Client . . . 43
5.2 Request and Result *getBlockHeader* . . . 44
5.3 Debug Information *getBlockHeader* . . . 45
5.4 Manipulate Response *getBlockHeader* . . . 45
5.5 Manipulated Response Error *getBlockHeader* . . . 46
5.6 Request and Result *getTransaction* . . . 46
5.7 Debug Information *getTransaction* . . . 46
5.8 Manipulate Response *getTransaction* . . . 47
5.9 Manipulated Response Error *getTransaction* . . . 47

1 Introduction

The number of IoT devices in the world is estimated to rise from 10 billion today to more than 30 billion by 2025. The role of such devices in everyday life is increasing rapidly. Amazon alone was able to sell more than 100 million Alexa devices by 2019. It is expected that many more use cases will be established within the next years. [3, 4]

Gartner, the world's leading research and advisory company, publishes a hype cycle for emerging technologies every year. In 2018 Gartner predicted that "Blockchain" and "IoT platform" will be on the plateau of productivity in 5 to 10 years. [5]

Bitcoin is the first and still the leading application realized based on the blockchain technology. It is one of the most dominant cryptocurrencies with around 300.000 transactions per day and a hash rate of around 130 EH/s[1] at the time of writing. Despite the big IoT device market, currently the connection between such devices and a blockchain is almost non-existent. [6]

The biggest challenge faced by the establishment of a connection between an IoT device and a blockchain is the operation of a blockchain client. Many low-performance IoT devices are neither capable of running a full node, nor a light node. The Trustless Incentivized Remote Node Network, in short IN3 (pronounced "in-cubed"), establishes a trustworthy and fast access to a blockchain. Therefore, it is a job enabler for a large number of IoT devices in terms of blockchain connectivity. IN3 is capable of supporting multiple chains, which enables the access to several blockchain data for IoT devices.

Currently, IN3 supports the verification of only Ethereum data. However, IN3 is able to support the verification of any blockchain due to its modularity. The goal of this thesis is to enable the verification of Bitcoin data in the IN3 protocol. To achieve this goal, fundamentals are briefly discussed, the developed concepts are presented and implemented. Finally, the implemented verification of Bitcoin in IN3 is demonstrated. The Bitcoin implementation will attract a wider audience and will allow actors to build more applications based on IN3. Therefore, billions of IoT devices in the world can be connected to one of the largest blockchain networks - Bitcoin.

[1] Exahash per second

2 Fundamentals

The following chapter deals with the necessary fundamentals of Bitcoin and IN3.

2.1 Bitcoin

This section briefly describes the mining process, the Difficulty Adjustment Period and the finality in Bitcoin.

2.1.1 Mining

"The process of trying to add a new block of transactions to the Bitcoin blockchain is called "mining". Miners are all participating in a network-wide competition, each trying to find a new block faster than everyone else. The first miner who finds a block broadcasts it across the network and other miners add it to their blockchain after verifying the block. Miners restart the mining-process after a new block is added to the blockchain to build on top of this block. As a result, the blockchain is constantly growing – one block every 10 minutes on average." [1]

How can miners find a block?

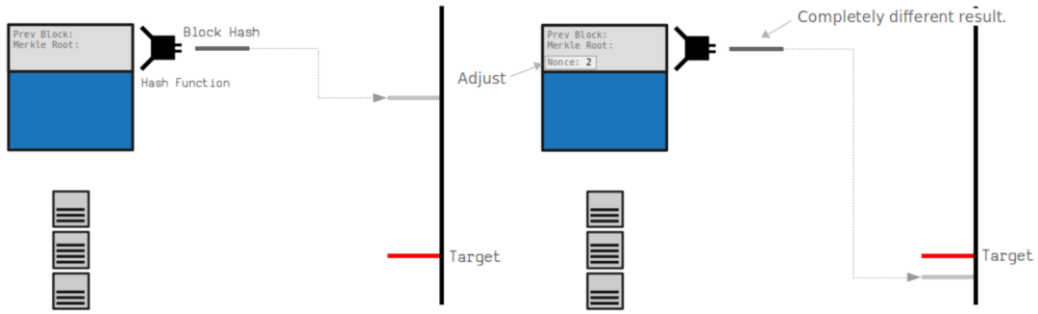


Figure 2.1: Bitcoin Mining [7]

Miners start the process by filling a candidate block with transactions from their memory pool. This is followed by the construction of a block header, which is a summary of all the data inside the block. Furthermore, the block header contains a reference to an already existing block in the Bitcoin blockchain. This reference is known as the parent hash. The mining process itself is hashing the constructed block header repeatedly, until the hash of the block header matches a certain target. One parameter is constantly being changed to receive a different hash for each repetition. The black line in figure 2.1 represents all possible hashes. For the mining process in Bitcoin the hash function SHA256 is being used, hence the black line represents 2^{256} hashes. Most calculated

hashes will not match the target (i.e. they will be greater than the target). One feature of such a hash function is that a result matching a certain target can only be produced by hashing different inputs again and again. The result cannot be determined in advance, nor does a pattern exist which produces a specific result. Eventually, one miner finds a hash that matches the target as shown on the right hand side in figure 2.1. The miner broadcasts its block across the network.

Upon receiving the new block, each node verifies its correctness and adds the new block to their blockchain. The competition for this block is over and nodes start creating a new candidate block with a reference to the recently found block. As a result, the blockchain is constantly growing. This process is referred to as **Proof-of-Work**.

Miners put their own special transaction at the top of a candidate block. This so-called coinbase transaction allows miners to send themselves a fixed amount of newly produced Bitcoins (6.25 BTC at the time of writing). The Bitcoins from a coinbase transaction can be spent once the block becomes 100 blocks deep in the longest blockchain. Besides the block reward, miners earn the transaction fees of the mined block: Total fees = sum(inputs) - sum(outputs). [7, 8, p. 244]

2.1.2 Difficulty Adjustment Period

"The white paper of Bitcoin specifies the block time as 10 minutes. Due to the fact that Bitcoin is a decentralized network that can be entered and exited by miners at any time, the computing power in the network constantly changes depending on the number of miners and their computing power. In order to still achieve an average block time of 10 minutes a mechanism to adjust the difficulty of finding a block is required: the difficulty." [1]

"The adjustment of the difficulty occurs every 2016 blocks - roughly every two weeks (which is one epoch/period). Bitcoin is a decentralized network and therefore there is no central authority which adjusts the difficulty. Instead, every miner compares the expected time to mine 2016 blocks (20160 minutes) with the actual time it took to mine the last 2016 blocks (using the timestamp of the first and last block). The difficulty increases when the blocks are mined faster than expected and vice versa. Although the computing power has increased heavily since the introduction of Bitcoin in 2009 the average block time is still 10 minutes due to this mechanism." [1]

Difficulty/Target

The difficulty is a number used to adjust the height of the target. The target is then used for the mining process or the verification of the Proof-of-Work of a mined block. The target can be calculated with the following formula: [10]

$$target = \frac{targetmax}{difficulty} \tag{2.1}$$

Targetmax =
0x00000000FFFF0000000000000000000000000000000000000000000000000000

Bits

The bits-section of a Bitcoin block header is used to store the target of that block header. It is not getting stored with absolute precision to save on space. The target being part of the block header offers the opportunity to verify a target by verifying the block header. This will be discussed in more detail later. [11]

The target can be extracted as follows:

Bits: 17110119_{16} (always 4 bytes)

- 1st byte: Total length of the target in hexadecimal form: 17_{16} = 23_{10} $\rightarrow$ the target has a total length of 23 bytes.
- 2nd - 4th byte: Are taken over into the target.
- The remaining bytes are filled with zeros to reach the total length (23 bytes in this example).

Resulting target: 1101190000000000000000000000000000000000000000

2.1.3 Finality

"In terms of Bitcoin, finality is the assurance or guarantee that a block and its included transactions will not be revoked once committed to the blockchain. Bitcoin uses a probabilistic finality in which the probability that a block will not be reverted increases as the block sinks deeper into the chain. The deeper the block, the more likely that the fork containing that block is the longest chain. After being 6 blocks deep into the Bitcoin blockchain it is very unlikely (but not impossible) for that block to be reverted." [1]

The finality of a block header can be seen as the basis for the verification of Bitcoin in the IN3 protocol. The details will be described in 3.1 followed by a security calculation of the finality of a block header in 3.3.2.

2.2 IN3

To enable smart devices to interact with a blockchain (e.g. sending a transaction), a blockchain client needs to run on this hardware. While devices with powerful hardware and good internet connection such as notebooks or desktop computers are able to run a full node, devices like smartphones or tablets with limited resources are only capable to run a light node. A full node processes and stores all data of a blockchain, whereas a light node only processes and stores the block headers. Many devices of the Internet of Things (IoT) have such severe limitations of computing power and connectivity that they are not even capable to run a light node. To enable such IoT devices to interact with a blockchain anyway, they can be connected to a remote node. The actual client (usually a full node) runs on powerful hardware and the IoT device is directly connected to that node to have access to required data. However, as a result, a big advantage of a decentralized network is undermined: not being forced to trust a single player. The IoT device has to trust the provided data by the remote node. Additionally, the remote node forms a single point of failure, hence there is no connection to blockchain data if the node fails. [2, 12]

The establishment of a secure and fast connection between IoT devices and a blockchain requires a client that is small enough (in terms of required resources) to run on a microcontroller. Furthermore, the client needs to be independent from a central instance.

2.2.1 Overview

"The Trustless Incentivized Remote Node Network, in short IN3 (pronounced "in-cubed"), makes it possible to establish a decentralized and secure network of remote nodes and clients which are able to verify and validate the results, enabling trustworthy and fast access to blockchain for many low-performance IoT, mobile devices, and web applications." [1]

The following problems that are preventing an IoT device to run a light node are solved by IN3:

- **Insufficient computing power and storage space**
  IN3 requires very little storage space (around 200 KB for the C implementation). Furthermore, IN3 needs little computing power because it does not need to synchronize, nor need to calculate a state.

- **Insufficient power supply**
  Light nodes should be constantly online to synchronize the blockchain. IN3 does not synchronize, hence can be switched off after usage to save on power. Therefore, an IN3 client can easily be powered by a battery without draining it too much.

- **No continuous connection to the internet**
  Light nodes need to keep synchronized or need to synchronize after a restart which could take a while with an unstable internet connection. Due to IN3 being a non-synchronizing client, the lack of a continuous internet connection is not a problem at all and it can be used instantly after a restart.

IN3 is a job enabler for many IoT devices that could otherwise not be connected to a blockchain due to their low computing power and unstable internet connection. [2, 12]

2.2.2 Architecture

The IN3 network consists of the following components:

- **IN3 Registry**
  The IN3 registry is a smart contract deployed on Ethereum. Every supported blockchain of IN3 has its own registry contract. Nodes that want to participate in the IN3 network must register themselves and pay a security deposit to the contract. Clients ask the nodes to send a node list from time to time. This list contains all currently registered nodes in the registry contract.

- **IN3 Node**
  An IN3 node is a full node of a supported blockchain. Nodes provide information and act as validators. Besides the actual data they provide proof data for the verification of the data itself. This allows the clients to fully verify the responses.

- **IN3 Client**
  An IN3 client requires minimal resources and can be installed e.g. on IoT devices. Clients request certain data from IN3 nodes and verify the response themselves by using the provided proof data.

- **Watchdogs**
  Watchdogs are special full nodes of the network which ensure that misbehavior of nodes is uncovered and penalized. IN3 nodes check the responses from other nodes, and therefore are passive Watchdogs. "Active Watchdogs" act like clients, but can verify the responses by themselves. In case a node gives an incorrect response, it gets convicted and loses its deposit.

The flow of a Remote Procedure Call (RPC) request can be seen in figure 2.2. Along with the actual RPC-request the client sends an optional list of validation nodes (A and C in the graph) to a randomly chosen node (node B). The node prepares the data and the proof data for the verification of the actual result on the client-side. Node B sends a request to the requested validation nodes to have the block hash signed. After receiving

all required signatures from the validation nodes (Validation Response in figure 2.2), node B sends a response to the client. This response contains the actual data, the proof data, and the signatures. Once the client received the response, it calculates the proofs and checks the provided signatures. The client has a crypto-economic assurance that the response from the server is correct, due to servers losing their security deposit upon signing a wrong blockhash. Maliciously acting server will be recognized by other servers (passive watchdogs) redirecting this information to the client or other nodes acting as active watchdog.

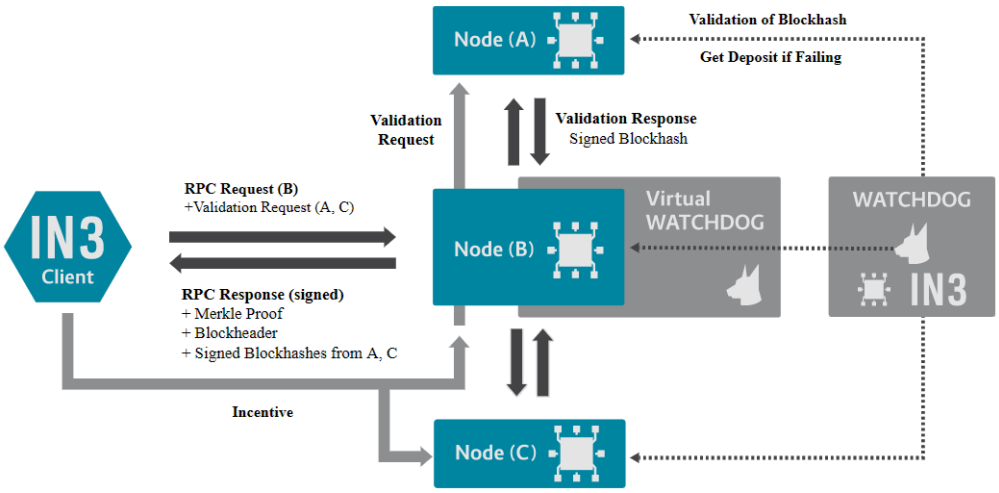


Figure 2.2: IN3 Architecture [13]

2.2.3 Incentivization

"As an incentive system for the return of verified responses, the node can request a payment. For this, however, the node must guarantee with its security deposit that the answer is correct. There are two strong incentives for the node to provide the correct response with high performance since it loses its deposit when a validator (wrong block hash) detects misbehavior and is eliminated from the registry, and receives a reward for this if it provides a correct response. If a client refuses payment after receiving the correctly validated information which it requested, it can be blacklisted or downgraded by the node so that it will no longer receive responses to its requests. If a node refuses to provide the information for no reason, it is blacklisted by the client in return or is at least downgraded in rating, which means that it may no longer receive any requests and therefore no remuneration in the future. The security deposit of the node has a decisive influence on how much trust is placed in it. When selecting the node, a client chooses those nodes that have a corresponding deposit (stake), depending on the security requirements (e.g. high value of a transaction). Conversely, nodes with a high deposit will also charge higher fees, so that a market with supply and demand for different security requirements will develop." [14]

At the time of writing the implementation of the incentive layer in the IN3 protocol is still under development.

2.2.4 Use Case: Smart Lock

"The rental of an e-bike is managed by a smart contract deployed on Ethereum. The lock is powered by the battery of the e-bike. It is equipped with a microcontroller to perform authorization checks and open the lock if necessary. Since the lock operates on the limited power of the e-bike, an internet connection is only established when needed for a check – therefore saving power in the remaining time. The installation of a blockchain client on the lock is necessary to establish a connection to the Ethereum blockchain. Installing a light node is not possible due to the limited resources (limited power supply, low computing power, no stable internet connection). Turning the light node on and off after each usage would indeed save electricity but would force the client to synchronize itself each time it comes back online – this would take too much time and requires a good internet connection. With an IN3 client running on the lock, a secure connection to the blockchain can be established at the required times only. Neither computing power is needed nor data is transferred in times when there is no rental process in action." [1]

3 Concept

The following chapter deals with the conceptional parts of the verification of Bitcoin data in the IN3 protocol. This includes the fundamentals of the verification, the necessary proofs, a security calculation and the concept of the on-chain conviction.

3.1 Verification

This section describes how the verification works in general and why it must be distinguished between two types of blocks. Eventually, the verification is described in detail for both types.

3.1.1 Design

The verification of Bitcoin data in the IN3 protocol uses the ideas of the Simplified Payment Verification (SPV) proposed in the Bitcoin white paper by Satoshi Nakamoto.

"It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it. As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network." [15]

The goal is the verification of the requested data on the client-side. The IN3 client does not store a chain of block header like a SPV client does. Instead, it is non-synchronizing and stateless. The communication between the client and the server happens via Remote Procedure Calls (RPC). Besides the requested data, the server provides proof-data enabling the verification of the actual data. This process is shown in figure 3.1. [16]

The provided proof-data depends on the request. The verification of the data itself requires several proofs, which are explained in 3.2. All supported methods and the verification of their result is described in 4.2.

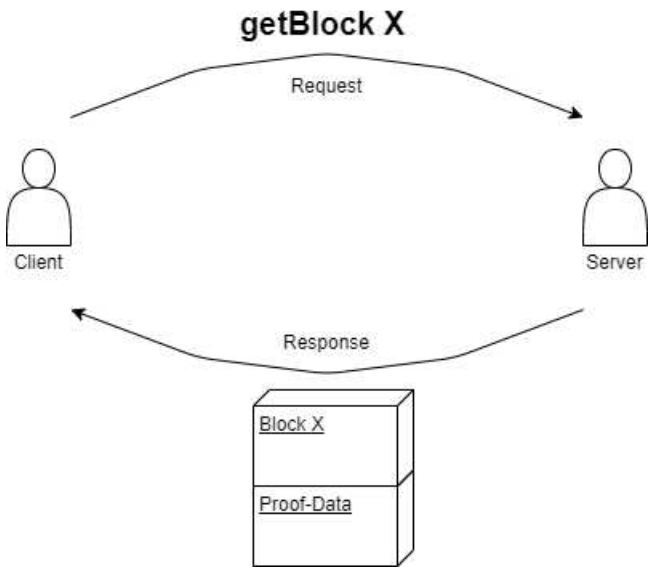


Figure 3.1: IN3 Request and Response

3.1.2 BIP34

In comparison to Ethereum, a Bitcoin block header does not contain the block number. The number of predecessors of a block is referred to as the height in Bitcoin. Therefore, the genesis block is at height 0 due to no predecessors and the block with 100 predecessors is at height 100. To verify the height of a block the links of blocks back to the genesis block have to be checked and counted. This verification requires the complete Bitcoin blockchain. Actors like an IN3 client that does not store the whole blockchain are not able to verify the height of a block. Gavin Andresen proposed a change to the Bitcoin protocol in 2012 to solve that problem. [1]

Bitcoin's community uses Bitcoin Improvement Proposals (BIP) as a quasi-standard to propose changes with a technical specification. Many changes that were implemented in the past are based on a BIP. [17]

Bitcoin Improvement Proposal 34 (BIP34) "introduces an upgrade path for versioned transactions and blocks. A unique value is added to newly produced coinbase transactions, and blocks are updated to version 2". After block number 227,835 (03/24/2013) all blocks must include the block height in their coinbase transaction. [18]

Although there is no block number in Bitcoin, the words "height" and "block number" will be used as synonyms from now on.

3.1.3 Blocks After BIP34

The verification of blocks after BIP34 relies on the finality of a block header. By proving the finality of a block header, the existence in the Bitcoin blockchain and its correctness can be verified. Having a verified block header allows verification of all parts of the block by performing the corresponding proofs. For example: a transaction can be verified by performing a Merkle proof. The security calculation in 3.3.2 shows how much security a certain number of finality headers provide.

3.1.4 Blocks Before BIP34

Relying on the finality does not work for blocks before BIP34 due to the following two problems:

- **Low mining power (low difficulty)**
  The total hash rate of the Bitcoin network was around 1-10 TH/s[2] in 2011, whereas today the total hash rate is around 130 EH/s. A single Antminer S9 SE is capable of running at 16 TH/s, which is more than the total hash rate back in 2011. Therefore, an attacker can easily mine a chain of fake-blocks with today's computing power. Hence, very old finality blocks does not provide any security at all. [6, 19]

- **Missing BIP34**
  The verification of the block number is an important part of the verification of bitcoin data in general. Since the block number is not part of the block header in Bitcoin, the client needs a different way to verify the block number to make sure that a requested block X really is block X. For every block after block number 227,835 the block number is part of the coinbase transaction due to BIP34. Blocks that are missing BIP34 need a different way of verifying the number of a block.

The verification of blocks before BIP34 relies on hard-coded checkpoints of hashes of bygone blocks on the client-side. The server provides the corresponding finality headers from a requested block up to the next checkpoint. By checking the link between the block headers the client is able to verify the existence and correctness of the requested block. The hash of the last provided finality header has to be equal to a checkpoint. The only way for an attacker to fool the client is by finding a different input that produces the same hash of a certain checkpoint (i.e. a hash collision). An attacker could provide a modified block header upon finding such a collision. The client would accept it because it would be able to verify the modified block header against a checkpoint. The security calculation in 3.3.1 shows the feasibility of such an attack.

[2] Terahash per second

The client has the opportunity to decide whether it wants to verify blocks before BIP34 or not. By turning on this option the checkpoints are included in the client software and for requests of blocks before BIP34 the server provides the corresponding finality headers to reach a certain checkpoint.

Creation of the checkpoints

The reason for using checkpoints is that it is not feasible for the client to save every single hash from the genesis block up to the introduction of BIP34. The checkpoints are hashes of bygone blocks and to save on space the checkpoints have a distance X. The larger this distance is, the smaller is the amount of checkpoints and the larger is the maximum number of necessary finality headers to reach a checkpoint. The maximum of required finality headers to reach a checkpoint is X-1. A large distance requires less storage space to save the checkpoints, however, the amount of finality headers per request will be very big. This results in a lot of data that has to be transferred per request. The following graph helps to decide on the distance between the checkpoints.

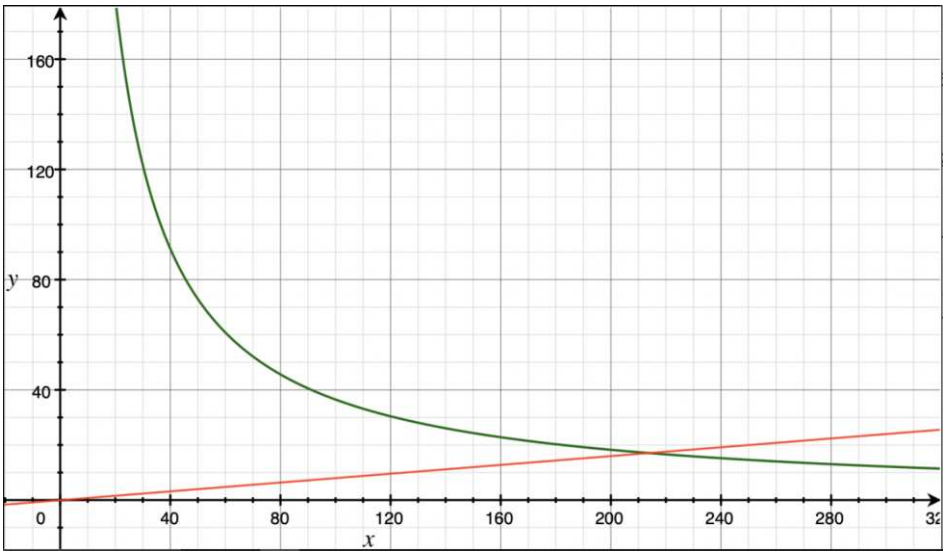


Figure 3.2: Distance Sweet Spot

- X-axis: Distance between checkpoints (blocks).
- Y-axis: Size in KB.
- Red: Maximum size of finality headers per request.
- Green: Size of record of checkpoints.

Although the lines cross at a distance of around 215, the distance of **200** can be seen as the sweet spot for the checkpoints. This means the record of checkpoints includes the hash of every 200th block of the Bitcoin blockchain starting with block 200. Storing the genesis block is not necessary, since a checkpoint always has to be in the future of a requested block. It takes 32 bytes to store a block hash. To save on space only the first 16 bytes are stored. Every hash starts with at least 4 bytes of zeros. To save even more

space the first 4 bytes can be removed as well. A client can simply prepend 4 bytes of zeros when working with a checkpoint. As a result, one checkpoint has a size of 12 bytes, therefore the record of checkpoints has a total size of 13680 bytes. Depending on the distance from a requested block to the next checkpoint a response includes a maximum of 199 finality headers which is a total of around 16 KB (15,920 bytes).

A checkpoint must always be in the future from the view of the requested block. The hash of block X-1 is part of block X (parent hash), but not vice versa. Hence, a checkpoint in the past does not provide any security. An attacker could simply modify block X to gain benefits and refer to block X-1 as the parent block. Only the Proof-of-Work has to be solved, which should not be too hard with today's computing power and the low difficulty in the early days of Bitcoin. Therefore, the verification of the existence and correctness of a block always requires finality headers **up to** the next checkpoint.

3.2 Proofs

A subset or all of these proofs are performed on the client-side to verify a server's response. The requested data determines which proofs have to be executed.

3.2.1 Target Proof

A verified target on the client-side is one of the most important things. A target can be used to verify the Proof-of-Work by simply checking if the hash of a block header is below the target. The acceptance of a wrong target must be prevented at all cost, otherwise an attacker could provide a target that is way bigger than the actual one. Hence, the attacker could mine a chain of fake-blocks with a lower difficulty which would lower the costs. The chain would be accepted on the client-side due to the hashes of the fake-blocks fulfill the requirement of the accepted fake-target.

The client maintains a cache with the number of a Difficulty Adjustment Period (DAP) mapped to the corresponding target. A target stays the same for the duration of one period. At the time of the release of the Bitcoin implementation this cache will be filled with default values. If a target is not yet part of the cache it needs to be verified first and added to the cache afterwards.

Using Finality

The target is part of a Bitcoin block header (bits-field). Hence, verifying a block header provides a verified target. This verification is based on the finality and requires a block header and a high number of finality headers. By proving the finality the existence and correctness of the block header and therefore the target can be verified. The greater the number of finality header, the higher is the probability of having a correct target. The

security calculation in 3.3.2 shows the provided security by a certain number of finality headers.

Additionally, a block number proof is necessary to map the verified target to the correct DAP. The number of the DAP can be gained with an integer division by 2016 of the verified block number (one period includes 2016 blocks).

Using Signatures

This approach is still under development, but shall be included for the sake of completeness.

Verifying a block header can also be achieved by requesting signatures of IN3 nodes. The client fetches the latest node list, selects one node which provides the block header (provider node) and several nodes that have to sign the hash of this block header (signatures nodes). The provider node manages to receive the signatures of all signature nodes. The response contains the block header and all requested signatures as well. The client can verify all signatures by using the node list which contains the public address of every node. This leads to a verified block header and therefore a verified target. The incentive for the nodes to act honestly is their deposit, which is lost upon acting maliciously.

Substantial Change

While setting up an IN3 client a value *maxDiff* is set, which determines the size of trusted target changes. This means that the client does not trust the changes of the target from a verified to a to be verified one when the changes are greater than the value of *maxDiff*. Whenever the client is not able to trust the changes of the target, it sends a special proofTarget-request to an IN3 server.

This method provides further data for the verification of substantial changes of the target on the client-side. Hence, the client can trust the substantial target changes. The data and the way verifying it will be described in 4.2.6 in more detail.

3.2.2 Finality Proof

This proof requires a block or block header and the headers of the following blocks (i.e. finality headers). The goal of the finality proof is to prove the existence and correctness of a Bitcoin block. In case a node is able to provide finality headers for a block, the probability of that block actually being part of the Bitcoin blockchain is very high.

The finality can simply be proven by checking the link between the finality headers. Every block header includes a parentHash-field which contains the hash of its predecessor.

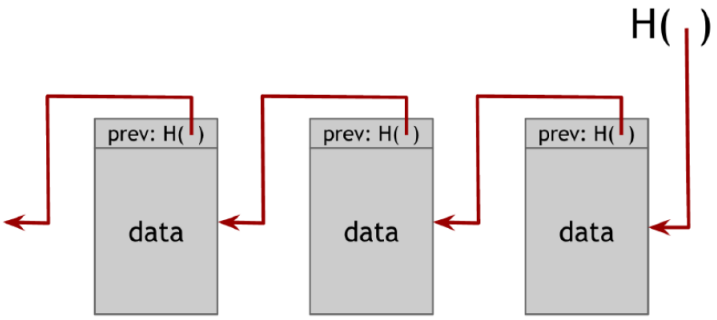


Figure 3.3: Successful Finality Proof [20]

The first link can be checked by hashing the first block header and checking this hash against the parent hash of the second block header. The second link can be check by hashing the second block header and checking this hash against the parent hash of the third block header. This process is shown in figure 3.3 and can be used to check the links of the whole chain of finality headers.

In case an attacker changes the data of a block header to gain benefits, the hash of this block header changes. Therefore, the link to the finality header is not correct anymore. The hash of the finality header changes as well. Hence, the link to the next finality header is also not correct anymore. A failed finality proof as shown in figure 3.4 signals that an attacker tried to modify the data of the block.

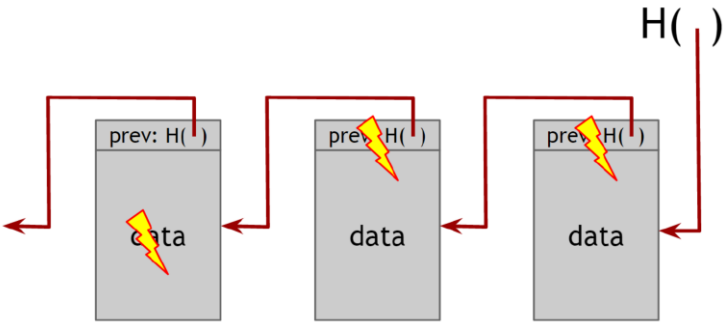


Figure 3.4: Failed Finality Proof [20]

The highest risk is a situation where a malicious node is able to provide a manipulated block header and finality headers that are probably valid but not actually valid (i.e. not part of the longest chain / chain of fake-blocks). The client would trust this data in case it has no other information to check against. The more finality headers, the greater the security, but also the more data that has to be transferred. The security calculation in 3.3.2 outlines the security of six finality headers.

Size

This proof requires:

- **Block header**
  A Bitcoin block header has a fixed size (80 bytes).
- **Finality headers**
  The size of the finality headers depends on the number (80 bytes · number).

The size of a finality proof increases linearly with the number of finality headers. A common procedure is to use six finality headers. Hence, the proof has a size of 560 bytes.

3.2.3 Transaction Proof

"All transactions of a Bitcoin block are stored in a Merkle tree. Every leaf node is labelled with the hash of a transaction, and every non-leaf node is labelled with the hash of the labels of its two child nodes. This results in one single hash (the Merkle root) which is part of the block header. Attempts to change or remove a leaf node after the block was mined (i.e. changing or removing a transaction) will not be possible since this will cause changes in the Merkle root, thereby changes in the block header and therefore changes in the hash of this block. By hashing the block header and checking this hash against the block hash such attempts will definitely be discovered. Having a verified block header and therefore a verified Merkle root allows to verify and prove the existence and correctness of a certain transaction." [1]

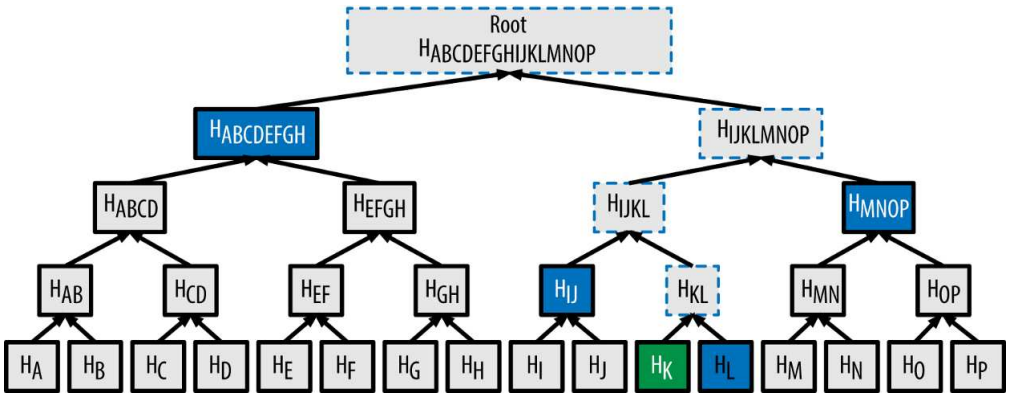


Figure 3.5: Merkle Tree [21]

"In order to verify the existence and correctness of transaction [K] SHA256 will be used to hash [K] twice to obtain H(K). For this example the Merkle proof data will contain the hashes H(L), H(IJ), H(MNOP) and H(ABCDEFGH). These hashes can be used to calculate the Merkle root as shown in 3.5. The hash of the next level can be calculated by concatenating the two hashes of the level below and then hashing this hash with

SHA256 twice. The index determines which of the hashes is on the right and which one on the left side for the concatenation (Hint: swapping the hashes will result in a completely different hash). When the calculated Merkle root appears to be equal to the one contained by the block header the existence and correctness of transaction [K] was proven. This can be done for every transaction of a block by simply hashing the transaction and then keep on hashing this result with the next hash from the Merkle proof data. The last hash must match the Merkle root." [1]

Size

This proof requires:

- **Block header**
  A Bitcoin block header has a fixed size (80 bytes).
- **Transaction**
  A transaction has a variable size. On average a size of 300 bytes. In case a transaction has many inputs and outputs, the size of the transaction increases heavily.
- **Merkle proof**
  The size of a Merkle proof depends on the number of transactions in the block. A block limit of 1 MB allows the storage of around 3,500 transactions per block. With this number of transaction a Merkle proof would include 12 hashes. Each hash has a size of 32 bytes, hence the Merkle proof has a total size of 384 bytes.

The size of a transaction proof depends on the size of the transaction itself and the number of transactions in the block. On average this proof has a size of **764 bytes**.

3.2.4 Block Number Proof

After the introduction of BIP34, all blocks must include number in their coinbase transaction. For all blocks after block number 227,835 the block number can be proven as follows:

1. Extract the block number out of the coinbase transaction:

   Coinbase transaction of block 624,692: [22]
   03348809041f4e8b5e7669702f7777772e6f6b65782e636f6d2ffabe6d6db388905
   769d4e3720b1e59081407ea75173ba3ed6137d32308591495198155ce0200000
   04204cb9a2a31601215b2ffbeaf1c4e00

This can be decoded: [23]

a) **03**: First byte signals the length of the block number (3 bytes).
b) **348809**: The block number in big endian - has to be converted to little endian.
c) **098834**: The block number in little endian - has to be converted to decimal.
d) **624692**: The actual block number.
e) **041f4e...**: The rest can be anything.

2. Prove the existence and correctness of the coinbase transaction:

   The existence and correctness of the coinbase transaction has to be verified to trust the extracted block number. This can be done with a Merkle proof. Therefore, the proof data has to contain a block header as well. The block header contains the Merkle root which is required for the Merkle proof.

Size

The size of a block number proof is similar to the size of a transaction proof (764 bytes on average). The coinbase transaction is a special transaction that has variable size as well. It is used to send the block reward and the transaction fees to the miner. The size of this proof depends on the size of the coinbase transaction and the number of transactions.

3.3 Security Calculation

The security calculation is split into two parts due to the different approach of the verification of blocks before BIP34 compared to the verification afterwards.

3.3.1 Blocks Before BIP34

The security of the verification of blocks before BIP34 is based on the fact that it is extremely hard to find input data whose leading 16 bytes of the hash are equal to a checkpoint's. Although a single checkpoint has the size of 12 bytes, it provides a security of 16 bytes. The first 4 bytes are zeros in every checkpoint, hence they can be removed to save on space. In other words: An attacker could manipulate a block, but to be accepted by the client, the hash of the last block of the chain of fake-blocks has to match the leading 16 bytes of a checkpoint. Finding a message that has a specific hash value is referred to as a **Preimage attack**. The resistance against such attacks is a crucial characteristic of a cryptographic hash function. Therefore, the approach of the verification of blocks before BIP34 offers a high degree of security through the design. [24]

This calculation uses the assumption that an attacker has to test 2^{128} possibilities on average to find such a hash. Since the output of a hash function can not be predicted, it might require the attacker way more or less tries to find such a hash. It might even be the first try with an unimaginable small probability. With a total hash rate of 120 EH/s in the Bitcoin network at the time of writing: [6]

$$\frac{2^{128}H}{120 * 10^{18}\frac{H}{s}} = 2,835,686,391,007,820,529s \tag{3.1}$$

It would take 89,919,025,590 years on average if the whole Bitcoin network with its current total hash rate would try to find such input data that results in a hash whose leading 16 bytes are equal to a checkpoint's.

Increased Security

Does the security increase if the requested block is further away from a checkpoint?

The further away a block is from the next checkpoint the more Proof-of-Work puzzles have to be solved by the attacker besides finding a matching hash to the checkpoint. Due to the low difficulty in the early days of Bitcoin (2009 - 2012) solving the Proof-of-Work puzzles is not a problem with today's computing power. With a total hash rate of around 9 TH/s in January 2012 it would take a few seconds or even parts of a second to find a matching hash today. Therefore, the security does not really increase with a further distance to a checkpoint because a few seconds do not have to be taken into account when compared with the 89 billion years it takes to calculate the final hash that has to match the checkpoint. [6]

Optimization

Saving 12 bytes per checkpoint is very secure with today's computing power as shown above. Removing two bytes causes a loss in security, but allows the reduction of the needed storage space or the data transferred per request. Reducing the size per checkpoint from 12 bytes to 10 bytes allows to:

- Reduce the size of all checkpoints from 13.68 KB to 11.40 KB, or
- keep the size of 13.68 KB for all checkpoints, therefore reduce the distance, hence reduce the maximum number of finality headers transferred per request.

With the reduced security it still would take the entire Bitcoin network 1.4 million years on average to find such input data whose hash matches a checkpoint.

The impact of reducing the size of all checkpoints is not as big as keeping the size and reducing the distance between the checkpoints to reduce the data transferred per request. While the record of checkpoints is only stored once, the finality headers will be part of every request. Keeping the size of 13.68 KB for the record of checkpoints allows the reduction of the distance to 167, hence a maximum of 166 finality headers per request (13.36 KB). A distance of 200 means a maximum of 15.92 KB finality headers per request. Therefore, the size of a single request can be reduced by up to 2.56 KB by reducing the distance.

At a first glance this does not seem to be that much, but it should be kept in mind that an IN3 client itself is only up to 300 KB in size. Additionally, reducing up to 2.56 KB per request reduces the size of 1,000 requests up to 2.56 MB and up to 128 MB for 50,000 requests - which is a lot for such a small client. The size of the record of checkpoints could also be increased to save more checkpoints, hence reducing the distance between the checkpoints. This would reduce the size of every request even more. The initial approach could be revised in the future to optimize it based on the numbers mentioned above.

3.3.2 Blocks After BIP34

The goal is to map the costs of a possible attack to the number of security providing finality headers. Although the result cannot be calculated exactly, it allows classification of the security.

A possible attack vector with the highest risk is a situation where an attacker could provide a manipulated block or block header (i.e. changing the data to gain benefits) and finality headers which fulfill the rules but are not actually valid (i.e. a chain of fake-blocks). The client would trust the manipulated data due to the fact that it would receive a positive result for the finality proof. In case the client requests six finality headers, the attacker has to calculate/mine a total of seven blocks - the finality headers plus the block itself. While based on assumptions and averages, the following calculation outlines the costs to do so.

Assuming that an attacker has 10% of the total mining power of the Bitcoin network, it would take around 100 minutes to mine one block due to the average block time of Bitcoin is 10 minutes. Therefore, mining seven blocks to perform the attack takes around 700 minutes. There are two things that have been taken into account for this calculation: Loss of mining reward and costs for the mining power.

While mining fake-blocks, the chances of earning block rewards are lost. Assuming that the attacker would have been able to mine seven real blocks. With a block reward of 6.25 BTC and a price of around $11,400 per Bitcoin at the time of writing, around **$498,750** of possible block rewards are lost. [6]

$$43.75BTC * \frac{\$11,400}{1BTC} = \$498,750 \tag{3.2}$$

Furthermore, the attacker needs to achieve 10% of the total mining power of the Bitcoin network. This would mean 12 EH/s with a total hash rate of 120 EH/s at the time of writing. Mining power can be achieved in two possible ways: Buying and operating mining hardware or renting the mining power. [6]

Bitmain is a company that sells various Bitcoin mining hardware. An Antminer S9 can be bought for around $100 and offers a hashing rate of 16 TH/s. To reach the required hashing power of 12 EH/s an attacker has to buy 750,000 of these miners, which would cost $75,000,000. The costs for electricity, storage room and cooling still needs to be added, making this option very expensive. [19]

Hashing power can also be rented online. The website nicehash.com is currently offering 1 PH/s[3] for 0.0098 BTC for 24 hours. Although the total available speed is only around 400 PH/s, this calculation assumes that it would be possible to rent 12 EH/s. Therefore, renting 12 EH/s for 24 hours would cost 117.6 BTC. Additionally, it assumes that the hashing power can be rented for 700 minutes only instead of 24 hours - which would be 48.6% of a day. [25]

$$117.6BTC * 0.486 * \frac{\$11,400}{1BTC} = \$651,510 \tag{3.3}$$

Taking both partial results into account, it can be assumed that six finality header provide a security of around **$1,150,260**. This number should be seen as a possible minimum. Due to the assumption that calculation is based on and the fact that it would be way harder to achieve such amount of hashing power in the real world, such an attack might be even more expensive. It should be noted that the result heavily depends on the current block reward and the price of Bitcoin.

Example

A rental car is equipped with an IN3 client running on a microcontroller to perform authorization checks and activate the ignition if necessary. The car is its own owner and it has a Bitcoin address to receive payments to lend itself to customers. Part of the authorization check is the verification of the existence and correctness of the payment using the IN3 client.

[3] Petahash per second

A customer sends the hash of the payment transaction to the car to be authorized in case the transaction gets verified. Assuming that a customer (Bob) runs a malicious IN3 node and the car randomly asks this node for the verification of the transaction. Bob could fool the car by creating a fake-transaction in a fake-block. To prove the correctness of the fake-transaction, Bob needs to calculate a chain of fake-blocks as well to prove the finality of the fake-block. In this case the car would authorize Bob because it was able to verify the transaction, even though the transaction is fake.

Bob would be able to use the car without having to pay for it, but performing such an attack is very expensive as shown above. This is what is meant by security in terms of dollar. Fooling the client in such a scenario is definitely not worth it since paying the actual fees for the car would be a far less than the cost for performing such an attack. Hence, IN3 clients can trust in the correctness of a transaction with a high probability when the value is less than $1,150,260 and the server is able to provide six finality headers for the block that contains that transaction. The higher the number of finality blocks, the higher the security due to increasing costs for the attack. The following graph 3.6 shows the security for the amount of finality headers based on the numbers used in the calculation.

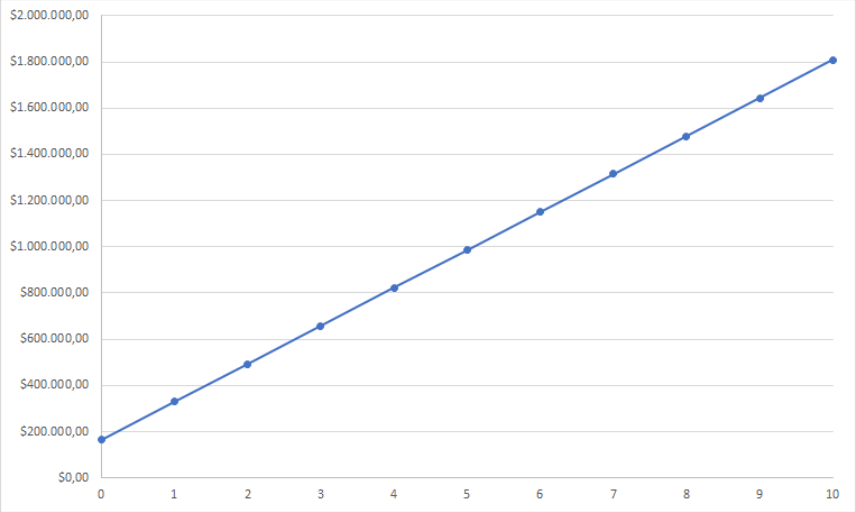


Figure 3.6: Security of Finality Headers

3.4 On-chain Conviction

Ethereum is often referred to as the world computer. "From a practical perspective, Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system's state change, along with a cryptocurrency called Ether to meter and constrain execution resource costs. (...) Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency Ether is both integral to and necessary for the operation of Ethereum, Ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer. (...) The term "smart contract" refers to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol - i.e., on the decentralized Ethereum world computer." [9, p. 1, 2, 127]

It can be assumed that most nodes of the IN3 network act honestly and provide correct data. Nevertheless, there might be some maliciously acting nodes as well which are trying to fool clients by providing manipulated data and even manipulated proof data. Such nodes have to be recognized and convicted. This can be done by a special authority (e.g. a court) in centralized networks.

The IN3 network is a decentralized network, hence maliciously acting nodes have to be convicted by a "decentralized court". The best way to realize the convict process in a decentralized manner is the implementation of that process in a **smart contract**. Nodes can charge other nodes in case they think that they are acting maliciously. The convict contract has to decide which node is the malicious node. It should be noted that malicious nodes can try to convict honest nodes as well. Therefore, the nodes (plaintiff and accused) play a "game" to prove who is acting honestly and who maliciously. The node which is able to provide a longer chain is most likely the honest node. The honest node has access to blocks from the public Bitcoin blockchain, whereas the malicious node needs to mine fake-blocks to play the game for a while. As long as the malicious node does not own 51% of the mining power, it will not be possible to create enough blocks to keep up with the longest chain in the public Bitcoin blockchain. Eventually, the malicious node gets recognized and convicted.

Longest Chain

The chain of blocks that took the most effort to be built is referred to as the longest chain. Due to the difficulty changes in the Bitcoin network, some blocks require more energy than others to be mined. Therefore, the longest chain is not necessarily the chain with the most blocks in it, but the one that required the most energy to be built. Although the chain on the right side in fig. 3.7 has more blocks in it, the chain on the left is the longer chain because it took more energy to be built.

A metric called chainwork is used to measure the length of a chain. The chainwork of one block is the total number of expected hashes to mine this block. A block with a greater difficulty (i.e. a lower target) has a higher amount of expected hashes to mine this block. Therefore, a greater difficulty adds more chainwork to a chain. By adding up the chainwork of each block in a chain, the chainwork of the complete chain can be calculated. Between two chains the one with a higher chainwork is the longer chain. [26]

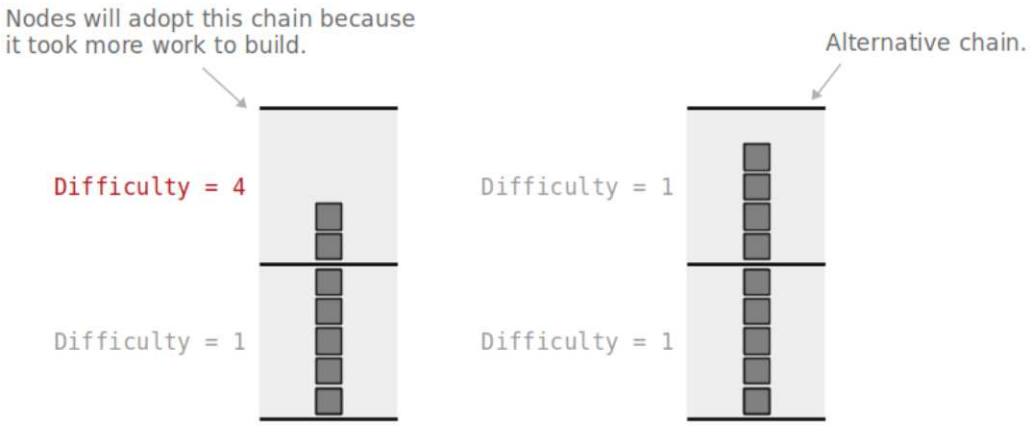


Figure 3.7: Bitcoin Longest Chain [26]

The Game

The actors of the game are two nodes: one that is acting honestly and one that is acting maliciously. The initial situation is the honest (called node B from now on) recognizing that another node (called node Y from now on) is acting maliciously (e.g. node Y provided a wrong block header). Node B starts the convict process with Node Y. The smart contract has to decide which node is the honest node. The malicious node loses its security deposit and is kicked out of the network.

The steps of the game in detail where node B is honest and node Y is malicious:

1. Node B starts the convict process: Submits the address of Node Y, the correct block header and the coinbase transaction plus Merkle proof for the block number proof.
2. Node Y recognized that another Node started a convict process with it and it has to submit some data as well: The wrong block header, the coinbase transaction plus Merkle proof for the block number proof and a block header referencing the wrong block header (i.e. a finality header).
3. The contract checks the chainwork of both chains and triggers an event with the information whose chainwork is greater.
4. Within a certain time period the node with the smaller chainwork has to submit a further block header to increase its chainwork.

5. The game will most likely be stuck in step three and four for a while. Eventually, the malicious node will not be able to create enough blocks to keep up with the honest node. Hence, the malicious node is not able to submit a further block header within the given time period.
6. The game ends with the removal of Node Y out of the network and a status that is set within the contract (like "convict_done") signaling that the process is over. The honest node can call the contract to collect the security deposit of Node Y.

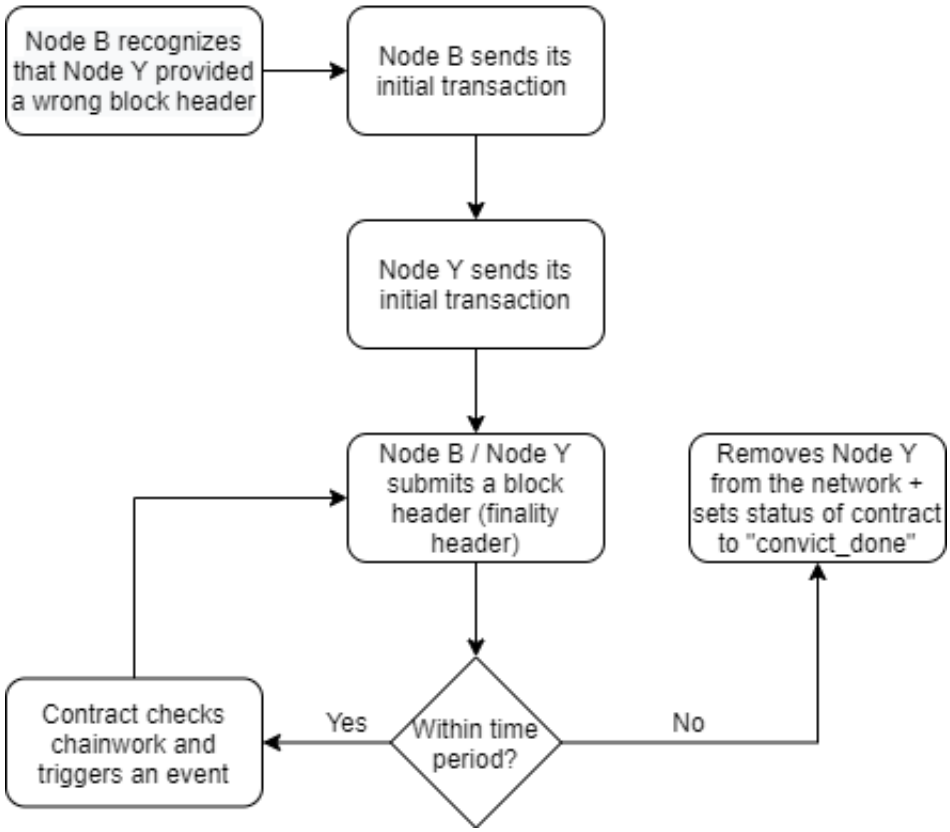


Figure 3.8: Game On-chain Conviction

More details about the process:

- The coinbase transaction and its Merkle proof only need to be part of the first transaction of both nodes. The block number for further block headers can be found out by incrementing the verified block number of the first block header. Transferring data in a smart contract is expensive, hence reducing the amount of data transferred reduces the costs.
- For now the on-chain conviction supports only blocks after BIP34. The block number is part of the coinbase transaction of such blocks. Verifying the block number of blocks before requires more effort (as described in 3.1.4) and might be supported in the future.

- The time period for a node to submit a new block header should be chosen to be large enough. One hour seems to be too little because it enables certain attack vectors. One of the nodes could flood the network with transactions for one hour and the transaction submitting a further block header of one node would not come through. Hence, an honest node could lose the game due to a crowded network. The time period should be a couple of days or even up to a week because it is not feasible to flood the network for such a long time.
- Basically everyone can start a convict process. Starting the process requires a security deposit so that the plaintiff and accused have something to lose - this prevents spam.

Balance

The balance of an address is the sum of all unspent transaction outputs (UTXOs) of that address. Without the on-chain conviction the verification of a balance is not feasible. While a node can prove the existence of certain transactions, it cannot prove that it has taken all UTXOs into account. Therefore, the client can verify its minimum balance, but the real balance might even be higher and cannot be verified.

Introducing the on-chain conviction enables the verification of a balance. In case a node has not taken all UTXOs into account, it can be convicted by another node. Hence, nodes always consider all UTXOs because otherwise they will definitely be convicted. Hence, clients are able to verify their balance.

4 Implementation

4.1 Architecture

The communication between the client and the server happens via Remote Procedure Calls (RPCs). The client sends one of the supported RPC-requests to the server. The server acts as a kind of proxy server and provides proof data besides the actual data, allowing the client to verify the response. The server fetches the Bitcoin data itself from a Bitcoin full node.

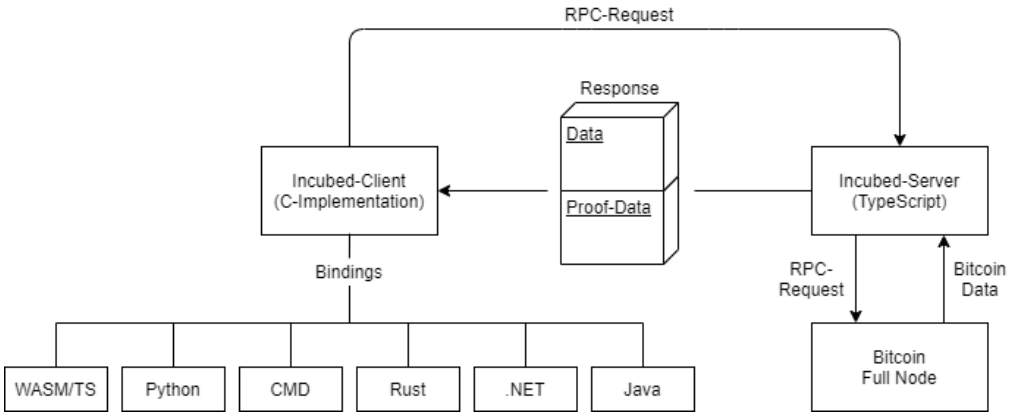


Figure 4.1: Architecture

"There are two options for an IN3 server to access Bitcoin data: Running a Bitcoin full node on the same machine as the IN3 server or connecting to a remote (trusted) Bitcoin full node. The operator of the IN3 server should make sure that he has access to correct Bitcoin data. This is in his own interest since he would lose his security deposit in case he delivers any wrong data. Therefore, operating a Bitcoin full node and the IN3 server on the same machine (or in the same trusted network) is highly recommended." [1]

A user has multiple options setting up and operating an IN3 client. The C implementation of the IN3 client forms the basis and is optimized to run on small embedded devices. The implementation contains several modules. By combining the required modules, only the required code is being generated, which reduces the requirements for flash and memory.

IN3 offers many bindings on top of the C implementation (see fig. 4.1). This allows to operate an IN3 client in many environments and enables many use cases. For example, IN3 can be used as a tool in Bash scripts or as a command-line utility. Furthermore, using the WASM version allows the use of IN3 in any JavaScript-runtime. [12]

4.2 RPC-Methods

Remote Procedure Calls (RPC) allow actors of a network to call processes located on another computer in the network. IN3 clients use RPC-requests to request certain data from an IN3 server. Bitcoin has a list of standard RPC-methods which every Bitcoin full node is able to respond to. There is a total of six requests out of the standard RPC-requests that IN3 servers support. The reason why there are only six supported requests is that there is a way to prove the actual result by adding proof data for these requests. Other requests whose response is not provable by adding proof data (e.g. a request about the current hash rate in the network) are not supported.

IN3 servers act like a normal Bitcoin full nodes for unsupported requests and respond without adding any additional data. Besides the supported RPC-methods, this chapter describes a special proofTarget-method which allows the verification of substantial changes of a target. This is not a standard RPC-method of Bitcoin, but a method that was created in this work. [27, 28]

A special in3-section is added to every request:

- **in3.finality**: Defines the number of finality headers.
- **in3.verification**: Defines the kind of proof the client is asking for (never or proof).
- **in3.preBIP34**: Defines if the client wants to verify blocks before BIP34 (true or false).

Every response contains a special in3-section containing a proof-object. This object contains parts or all of the following properties (depending on the request):

- **block**: A hex string with 80 bytes representing the block header.
- **final**: The finality headers, which are hex-coded bytes of the following headers (80 bytes each) concatenated. The number depends on the requested finality in the in3-section of the request.
- **txIndex**: Index of the transaction. This is necessary to perform a Merkle proof.
- **merkleProof**: The Merkle proof for the requested transaction proving the existence and correctness of the transaction.
- **cbtx**: The serialized coinbase transaction of the block. This is needed to get a verified block number.
- **cbtxMerkleProof**: The Merkle proof of the coinbase transaction proving the existence and correctness of the coinbase transaction.
- **height**: The height of the block.

The description of the methods will be in the following pattern: Parameters, result, proof data and verification. Two detailed example requests are shown in Chapter 5.

4.2.1 getBlock

Returns data of a block for a given block hash. The returned level of details can be configured in the request.

Parameters

Besides the in3-section this request contains:

- **blockhash**: The hash of the requested block.
- **verbosity**: Defines the returned level of details (0, 1 or 2).

Result

The result depends on the verbosity:

- **0**: A serialized string which represents the hex-encoded block data.
- **1**: The block as a JSON-object.
- **2**: The block as a JSON-object with detailed information about each transaction included in that block.

Proof Data and Verification

The proof data depends on the height of the block (before or after BIP34) and the value of the property in3.preBIP34 which defines whether the client wants to verify blocks before BIP34 or not.

- **Blocks before BIP34 and in3.preBIP34=false**
  The proof data includes the finality headers of the requested block. These finality headers provide no security at all as described in 3.1.4. Therefore, the verification of such blocks is not possible with in3.preBIP34 being set to false.

- **Blocks before BIP34 and in3.preBIP34=true**
  The proof data includes the finality headers up to the next checkpoint and the height of the requested block. The verification of such blocks can be done as described in 3.1.4.

- **Blocks after BIP34**
  The proof data includes the finality headers, the coinbase transaction and the Merkle proof for the coinbase transaction. Proving the finality of the requested block proves the existence and correctness of that block. The block number is extracted from the coinbase transaction and the Merkle proof allows the verification of that transaction and thereby the extracted block number.

4.2.2 getBlockHeader

Returns data of a block header for a given block hash. The returned level of details can be configured in the request.

Parameters

Besides the in3-section this request contains:

- **blockhash**: The hash of the requested block header.
- **verbosity**: Defines the returned level of details (0 or 1).

Result

The result depends on the verbosity:

- **0**: A 80 bytes hex string representing the block header.
- **1**: The block header as a JSON-object.

Proof Data and Verification

The proof data depends on the height of the block (before or after BIP34) and the value of the property in3.preBIP34 which defines whether the client wants to verify blocks before BIP34 or not.

- **Blocks before BIP34 and in3.preBIP34=false**
  The proof data includes the finality headers of the requested block header. These finality headers provide no security at all as described in 3.1.4. Therefore, the verification of such block headers is not possible with in3.preBIP34 being set to false.

- **Blocks before BIP34 and in3.preBIP34=true**
  The proof data includes the finality headers up to the next checkpoint and the height of the requested block header. The verification of such block header can be done as described in 3.1.4.

- **Blocks after BIP34**
  The proof data includes the finality headers, the coinbase transaction and the Merkle proof for the coinbase transaction. Proving the finality of the requested block header proves the existence and correctness of that block. The block number is extracted from the coinbase transaction and the Merkle proof allows the verification of that transaction and thereby the extracted block number.

4.2.3 getTransaction

Returns the transaction data for a given transaction ID. The returned level of details can be configured in the request.

Parameter

Besides the in3-section this request contains:

- **txid**: The ID of the requested transaction.
- **verbosity**: Defines the returned level of details (0 or 1).
- **blockhash**: The hash of the block that contains the requested transaction.

Result

The result depends on the verbosity:

- **0**: A serialized string which represents the hex-encoded transaction data.
- **1**: The transaction as a JSON-object.

Proof Data and Verification

The proof data depends on the height of the block (before or after BIP34) and the value of the property in3.preBIP34 which defines whether the client wants to verify blocks before BIP34 or not.

- **Blocks before BIP34 and in3.preBIP34=false**
  The proof data includes the block header of the block that contains the requested transaction, the finality headers, the index of the transaction and the Merkle proof for the transaction. These finality headers provide no security at all as described in 3.1.4. Therefore, the verification of such transactions is not possible with in3.preBIP34 being set to false.

- **Blocks before BIP34 and in3.preBIP34=true**
  The proof data includes the block header and the height of the block that contains the requested transaction, the finality headers up to the next checkpoints, the index of the requested transaction and the Merkle proof for the requested transaction. The verification of such transactions can be done by the verification of the block header as described in 3.1.4 and the verification of the transaction itself by performing a Merkle proof afterwards.

- **Blocks after BIP34**
  The proof data includes the block header of the block that contains the requested transaction, the finality headers, the index of the requested transaction, the Merkle proof for the requested transaction, the coinbase transaction of the block that contains the requested transaction and the Merkle proof for the coinbase transaction. The block header can be verified by performing a finality proof. Having a verified block header allows the verification of the transaction itself with a Merkle proof. The block number is extracted from the coinbase transaction and the Merkle proof allows the verification of the coinbase transaction and thereby the extracted block number.

4.2.4 getBlockCount/getBestBlockHash

Although these are two different requests, they are combined in this section due to similarity. The only little difference is the result, everything else is the same for both requests.

Parameter

Neither of the requests have parameters.

Result

While getBlockCount returns the **number of blocks** in the longest blockchain, getBestBlockHash returns the **hash** of the best block in the longest blockchain. It is not possible to prove the finality of the latest block because this block is not final yet. Therefore, latest block means actual latest block minus number of finality headers.

The finality can be set to 0 in the request to get the actual latest block number. This block is not final and could no longer be part of the blockchain later on due to the possibility of a fork. Additionally, there may already be a newer block that the server does not yet know about due to latency in the network.

Proof Data and Verification

The proof data includes the block header of the "latest" block (actual latest block minus number of finality headers), the finality headers, the coinbase transaction and the Merkle proof for the coinbase transaction. Proving the finality of the block header proves the existence and correctness of that block. The block number is extracted from the coinbase transaction and the Merkle proof allows the verification of that transaction and thereby the extracted block number. The result of getBlockCount can be verified by checking it against the verified block number. The result of getBestBlockHash can be verified by hashing the verified block header.

4.2.5 getDifficulty

The Proof-of-Work difficulty is being returned as a multiple of the minimum difficulty.

Parameter

Besides the in3-section this request contains:

- **blocknumber**: This property can be a certain block number for the difficulty of a specific block **OR** for the difficulty of the latest block it must be latest, earliest or pending.

Leaving the parameters empty requests the difficulty of the latest block as well.

Result

The result depends on the requested block number:

- **blocknumber is a certain number**: The difficulty of this specific block.
- **blocknumber is one of the strings mentioned above or empty**: The difficulty of the latest block.

It is not possible to prove the finality of the latest block because this block is not final yet. Therefore, latest block means actual latest block minus number of finality headers.

Proof Data and Verification

The proof data depends on the height of the block (before or after BIP34) and the value of the property in3.preBIP34 which defines whether the client wants to verify blocks before BIP34 or not.

- **Blocks before BIP34 and in3.preBIP34=false**
  The proof data includes the block header of the requested block number and the finality headers. These finality headers provide no security at all as described in 3.1.4. Therefore, the verification of the requested difficulty is not possible with in3.preBIP34 being set to false.

- **Blocks before BIP34 and in3.preBIP34=true**
  The proof data includes the block header of the requested block number, the finality headers up to the next checkpoint and the height. The verification of the difficulty can be achieved by the verification of the block header as described in 3.1.4.

- **Blocks after BIP34**
  The proof data includes the block header of the requested block number, the finality headers, the coinbase transaction and the Merkle proof for the coinbase transaction. The block header and the finality headers are used to perform a finality proof. The block number can be verified by extracting it out of the coinbase transaction and proving the existence and correctness of that transaction with a Merkle proof.

4.2.6 proofTarget

While setting up an IN3 client a value *maxDiff* is set, which determines the size of trusted target changes. Whenever the client does not trust the changes of the target, this method is being called. It returns a path of Difficulty Adjustment Periods (DAP) to verify the target step by step to trust the changes. Figure 4.2 visualizes the usage of that method.

Parameter

Besides the in3-section this request contains:

- **target_dap**: The number of the DAP whose target needs to be verified.
- **verified_dap**: The number of the closest DAP whose target is already verified.
- **max_diff**: The client's trusted size of target changes between two DAPs.
- **max_dap**: The max. distance between two DAPs in the path.
- **limit**: The max. number of DAPs in the path.

Result

This method returns a path of DAPs from the verified dap to the target dap which fulfills the conditions of max_diff, max_dap and limit. Each DAP of the path contains the first block header of that DAP with corresponding proof data. The verification of the block header provides a verified target for this DAP.

Proof Data and Verification

The proof data depends on the height of the block (before or after BIP34) and the value of the property in3.preBIP34 which defines whether the client wants to verify blocks before BIP34 or not.

- **Blocks before BIP34 and in3.preBIP34=false**
  Each DAP of the path includes the number of the DAP, the block header of the first block of the DAP and the finality headers. These finality headers provide no security at all as described in 3.1.4. Therefore, the verification of the block header and thereby the target is not possible with in3.preBIP34 being set to false.

- **Blocks before BIP34 and in3.preBIP34=true**
  Each DAP of the path includes the number of the DAP, the block header of the first block of the DAP, the finality headers up to the next checkpoint and the height of the block header. The verification of the block header can be achieved as described in 3.1.4. As already mentioned, having a verified block header provides a verified target for this DAP.

- **Blocks after BIP34**
  Each DAP of the path includes the number of the DAP, the block header of the first block of the DAP, the finality headers, the coinbase transaction and the Merkle proof for the coinbase transaction. Again, the goal is the verification of the block header. This time relying on the security of the finality headers. Besides the block header verification, the block number has to be verified. This can be done by extracting the block number out of the coinbase transaction and proving the existence and correctness of this transaction using a Merkle proof.

A path can contain both type of blocks, before and after BIP34. Therefore, the proof data and the verification can look different within one response/path. If the verification of blocks before BIP34 is disabled, then it might happen that the part of the path with blocks before BIP34 is not verifiable, whereas the part with blocks after BIP34 is.

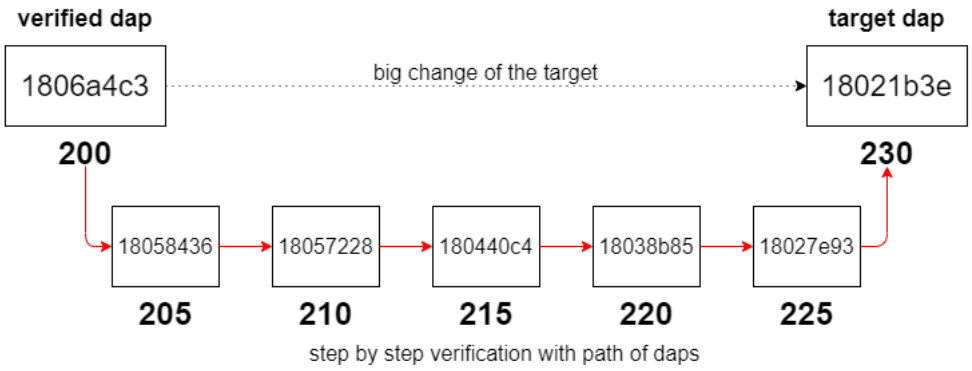


Figure 4.2: Visualization of proofTarget

4.3 Cache

A cache is a part of the memory with a very short access time. It stores recently and frequently used data. [29]

4.3.1 Server-Cache

Parts of the data that is used to respond to requests from a client is being cached. There is a chance for the server to find the necessary data for further responses in its cache which saves on computing power and responding time.

The current implementation of the cache in the server is a simple map with a block hash and a block number pointing at the same cache object. Therefore, each cache entry has two keys to be identified. It is not feasible to cache the whole data of every block. A cache object stores the most important data only: the block height, the block hash, the block header, the transaction IDs and the coinbase transaction. The creation of a cache object does not require all five properties at once, but it can be created with a subset of the data instead. It can be completed later on with the corresponding data.

Most requests can be responded with the data in the cache. For example: The block number proof requires the coinbase transaction and a Merkle proof for this transaction which can be created with the transaction IDs. At the time of writing the Bitcoin blockchain consists of around 653,000 blocks. Storing a cache object for every block would require around **125 GB** of storage space.



Figure 4.3: Demonstration of the Cache

The best way to demonstrate the improvement in terms of response time by using the cache is by using an IN3 server locally and a software to send a request to the server (e.g. Postman). Although it runs locally, it is connected to a Bitcoin full node to fetch Bitcoin data. Fetching the necessary Bitcoin data from the full node takes the most time in this case because the Bitcoin full node and the IN3 server are not running on the same machine. A getBlockHeader-request is being sent to the server twice. The responding time of the second request should be significantly lower due to the fact that the necessary data is in the cache and fetching it again from the Bitcoin full node is not

necessary. The numbers in figure 4.3 show the improvement in terms of responding time. These numbers can deviate in a production environment.

The current implementation of the cache can be considered as the first version of the cache. As shown above, improvements in terms of responding time were achieved. Nevertheless, the current implementation is not the perfect solution. A new implementation of the cache using a popular library is planned for the future. Two libraries that are worth considering are **Redis**[4] and **Node-Cache**[5]. Such libraries allow to manage the storage space in a better way and offer more functionality in general. For example, an external disk can be used as storage space and data that has not been used in a long time can be replaced with new data in case the limit of the storage space is reached.

4.3.2 Client-Cache

The cache of the client contains two things: The node list and a map between Difficulty Adjustment Periods (DAP) and their target.

Node List

The node list contains the public addresses of all registered IN3 nodes. The node list can be fetched from one of the boot nodes after setting up an IN3 client. These are highly available nodes whose addresses are stored in the client per default. Caching the node list makes absolutely sense to prevent fetching the list again for every request. Nevertheless, updating the node list from time to time is recommended due to changes that might happen to the list.

Target

The target of bygone Difficulty Adjustment Periods (DAP) remain unchanged in the future. Therefore, it is possible to fill the target-cache with default values already. While releasing the Bitcoin implementation, the target-cache will be filled with bygone DAPs and their target. Therefore, the client has access to verified targets without having sent a single request. An entry has a size of 6 bytes and contains the number of the DAP and the corresponding target of that DAP. To save on space, only every fifth DAP is getting stored in the initial cache starting at the tenth DAP (10, 15, 20, ...).

Unverified targets can be verified with the help of the cache (access to already verified targets), the usage of the target proof as described in 3.2.1 and the usage of the proofTarget-method as described in 4.2.6. Newly verified targets are getting stored in the cache for further usage.

[4] https://redis.io/
[5] https://www.npmjs.com/package/node-cache

Example of two cache entries: **006962fa041a**$_{16}$ and **00b9c14d1318**$_{16}$.

- The first two bytes are representing the number of the DAP:
  $0069_{16} = 105_{10}$ and $00b9_{16} = 185_{10}$.
- The following 4 bytes are representing the target of the corresponding DAP:
  $62fa041a_{16}$ and $c14d1318_{16}$.

The target is being stored in a compact way using the format of the bits-section of a Bitcoin block header. Section 2.1.2 explained the steps to gain the actual target out of this format.

4.4 Tests

The creation and maintenance of test scripts might take a little more effort than manual testing, but allows to automatically test certain cases over and over again and increases the overall test coverage.

4.4.1 Server-Tests

The IN3 server is implemented in TypeScript/JavaScript. A JavaScript framework running on Node.js is being used for automated testing in the server: **Mocha**[6]. Tests are created for every unit of the code, hence this method of testing is referred to as unit testing.

Each test includes three sections:

- **request**: The request with its parameters.
- **mock_responses**: All requests and responses that the IN3 server sends to and receives from the Bitcoin full node.
- **expected_result**: The expected result of the request.

The execution and the management of the tests is done by a so-called test runner, which is a piece of code. The test runner starts by sending the request-section of the first test to the server. This request is being handled as a normal request on the server-side. The server communicates with the Bitcoin full node to fetch the requested data. This sequence of communication has to be the same as defined in the test (mock_responses-section). The server sends a response for the request which is handled by the test runner. It compares the response from the server with the response defined in the test (expected_result-section). The test is successful if the response is equal to the expected

[6] https://mochajs.org/

response. Otherwise, the test runner throws an error. This process is executed for every test.

Tests that are testing error cases are referred to as negative tests. The request-section of such tests is provoking an error, for example by sending a wrong parameter. These test will still be successful, although the response is an error because the error is the expected result.

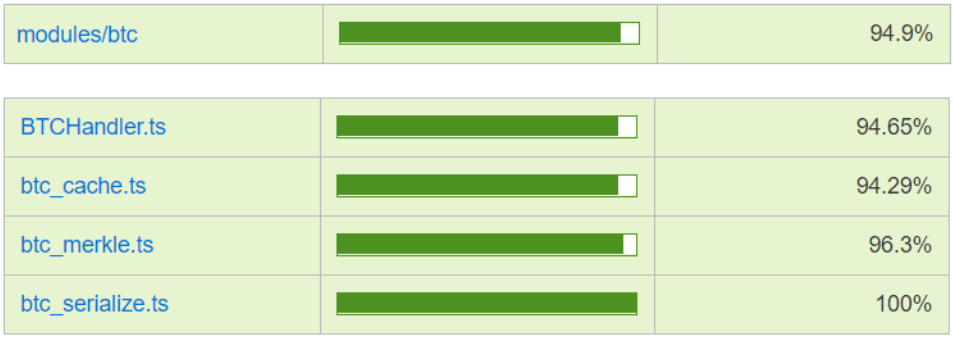


Figure 4.4: IN3 Server Code Coverage [30]

At the time of writing the Bitcoin implementation of the server has 74 tests generating a code coverage of around 95%. The code coverage can be pushed up to 100% in the future by adding specific tests that are covering the missing lines of code. Fig. 4.4 shows the files that are in the Bitcoin module of the server.

Every test has a short description showing what that test is about. The following descriptions should give an overview about the current tests:

- BTC-Tests - getBestBlockHash: call with no finality (finality = 0)
- BTC-Tests - getBestBlockHash: call with wrong amount of parameters
- BTC-Tests - getBlockHeader: ask for finality headers that are not existing yet
- BTC-Tests - getBlockHeader: get a block header before BIP-34 (block 197,881)
- BTC-Tests - getDifficulty: without proof (block 641,601)
- BTC-Tests - getTransaction: use non-existing block hash

4.4.2 Client-Tests

The C implementation of the IN3 client forms the basis as described in 4.1. Therefore, this section is about the tests of the C implementation only. For JSON-tests the same framework as for the server is being used: **Mocha**. Additionally, the C client uses a framework for units tests directly in C: **Unity**[7]. The other bindings are using different frameworks depending on the programming language.

[7] http://www.throwtheswitch.org/unity

The procedure of the actual testing is equal to the one described in the tests of the server. Test with the Unity-framework have a different syntax, but the procedure is similar to the Mocha-framework.

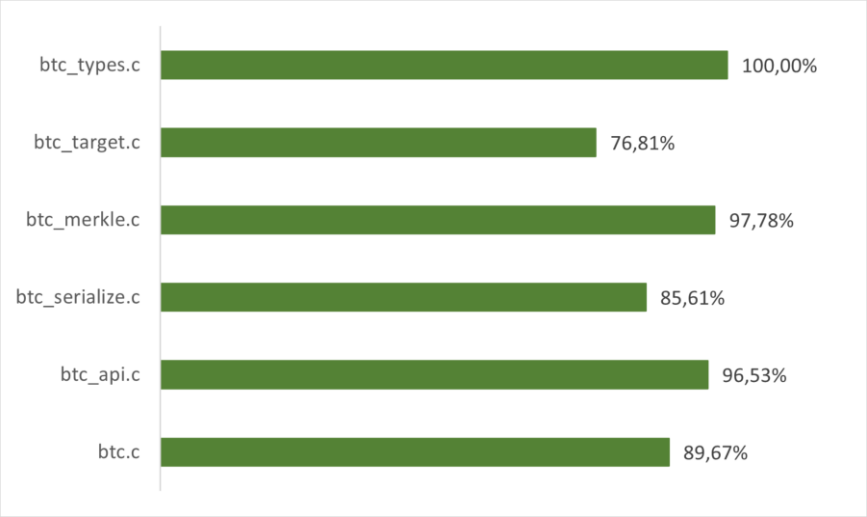


Figure 4.5: IN3 Client Code Coverage

At the time of writing the Bitcoin implementation of the client has a code coverage around 91%. Currently, especially tests for negative cases are still missing. By adding such tests the coverage can be pushed to 100% in the future. Figure 4.5 shows the files that the Bitcoin module of the client consists of.

Furthermore, the JSON-tests are using a popular testing technique: **Fuzzing**. It can be defined as: "A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines." [31, p. 58-62]

4.5 On-chain Conviction

The convict process described in 3.4 needs to be implemented in solidity. This section describes the first function that was implemented as part of this work. It allows the verification of the Proof-of-Work of a block header.

General procedure of the verification of the Proof-of-Work of a block header:

- **Extract Bits**
  A Bitcoin block header has a fixed size of 80 bytes. The bits-section has a fixed size of four bytes and starts at the 72nd byte of the block header.

- **Convert Bits**
  The first byte of the Bits is the total number of bytes of the target (hex). The other bytes are taken over into the target and the remaining bytes are filled with zeros (to reach the total number of bytes). More details were explained in 2.1.2.
  Example: 17110119 $\rightarrow$ 1101190000000000000000000000000000000000000000

- **Compute Hash**
  The hash of the block header is computed by hashing the block header with SHA256 twice.

- **Check Hash against Target**
  Simply check if the hash of the block header is lower than the target. If yes, the Proof-of-Work is verified for this block header. If not, the hash of the block header does not fulfill the requirements of the target.

```
function verifyHeader(bytes memory header) public pure returns (bytes32 hash, bytes32 target, bool result) {
    bytes memory bytesHash = swap(doubleHash(header)); // get hash and swap order of bytes

    // convert to bytes32
    assembly {
        hash :=mload(add(add(bytesHash, 0x20), 0))
    }

    bytes memory headerBits = swap(getBits(header)); // get bits out of header and swap order of bytes

    uint8 length = uint8(headerBits[28]); // bits are at [28][29][30][31] of header bits
                                          // first byte of the bits is the prefix (at [28])
                                          // convert hex to number

    // put the last 3 bytes of "headerBits" into "target"
    assembly {
        target :=mload(add(add(headerBits, 0x20), 29))
    }

    target = target >> 8 * (32 - length); // shift the bytes to the correct spot
    result = (hash < target); // hash below target: true, othwise: false
}
```

Figure 4.6: Soldity Code *verifyHeader*

Parts of the implementation of the procedure in solidity is shown in fig. 4.6. The main function *verifyHeader* has one parameter: The block header. It returns the hash of the block header, the target and a boolean value signaling if the hash is lower than the target. This is only the first approach and this function will most likely return a boolean value only in the future.

Functions that are used by the verifyHeader-function:

- **doubleHash**
  Parameter: A block header.
  Returns: The hash of the block header (SHA256 twice) as a bytes32 array.

- **swap**
  Parameter: A bytes32 array.
  Returns: A bytes array with reversed bytes.

- **getBits**
  Parameter: A block header.
  Returns: The bits-section of the block header as a bytes4 array.

Procedure of the verifyHeader-function:

1. Get the hash of the block header and reverse the order of bytes.
2. Convert the hash to a bytes32 array (swap returns a bytes array).
3. Extract the Bits out of the block header and reverse the order of bytes.
4. Determine the total number of bytes of the target.
5. Take the last three bytes of the Bits over into the target.
6. Fill the remaining bytes with zeros by shifting the bytes.
7. Check the hash against the target.
8. Return true if the hash is lower than the target, otherwise false.

The implementation of the verifyHeader-function can be considered as the first approach of the implementation of the on-chain conviction. Being able to verify the Proof-of-Work of a block header is a huge step. Nevertheless, many more functions are still missing. The next step would be the implementation of the block number proof. This requires the extraction of the block number out of a coinbase transaction and the verification of its correctness by performing a Merkle proof. The implementation of the game described in 3.4 will most likely require the most effort. The complete implementation and documentation of the on-chain conviction based on the presented concept can be considered as future work of this thesis.

5 Demonstration

The following chapter demonstrates the verification of Bitcoin in the IN3 protocol. This includes the setup, two requests as examples and a description for further demonstration. The most important lines of code are displayed within the sections.

5.1 Setup

Client Setup

IN3 offers several bindings to use the IN3 client in different environments. For this demonstration a JavaScript-runtime (Node.js[8]) and the IN3 WASM client are used.

```
const IN3 = require("in3-wasm")

const in3 = new IN3({
    "chainId": "0x99",
    "finality": 8,
    "maxAttempts": 4,
    "debug": true,
    "maxDAP": 5,
    "maxDiff": 15,
    "nodes": {
        "0x99": {
            "nodeList": [{
                    "address": "0x1234567890123456789012345678901234567890",
                    "url": "https://in3-v2.slock.it/btc/nd-2",
                    "props": "0xFFFF"
                }
            ]
        }
    }
})
```

Figure 5.1: Setup WASM Client

The properties have the following meaning:

- **chainId**: 0x99 = Bitcoin.
- **finality**: Number of finality headers.
- **maxAttempts**: Max. number of request attempts.
- **debug**: Show debug information.
- **maxDAP**: Max. distance between DAPs (*proofTarget*).
- **maxDiff**: Max. difference between two targets (*proofTarget*).
- **nodes - 0x99 - nodeList**: List of Bitcoin nodes (just 1 node here).

[8] https://nodejs.org/en/

- **address**: Public address of the node.
- **url**: URL of the node.
- **props**: A bitmask defining the capabilities of the node (0xFFFF = everything).

At the time of writing signing for Bitcoin is not implemented yet. Therefore, the property *address* does not matter and can be anything (currently it is "0x123456780..."). This property will be the public address of the node once implemented.

The in3-object is now configured and ready to send requests to the server.

Server Setup

For this demonstration an IN3 server operated by Blockchains LLC is used. This server has the latest version installed and supports the verification of Bitcoin data, hence it provides proof data. The URL of the server is *https://in3-v2.slock.it/btc/nd-2*. This is the only server in the configuration of the client as shown in fig. 5.1. Therefore, all requests will be sent to this particular node.

5.2 Examples

Request Block Header

This request returns the block header as a JSON-object for a given block hash.

```
async function sendRequest() {
    const blockheader = await in3.btc.getBlockHeader("000000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7")
    console.log(blockheader)
}

sendRequest().catch(console.error)

// Result:

{ "hash": "000000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7",
"confirmations": 27682,
"height": 625000,
"version": 1073733632,
"versionHex": "3fffe000",
"merkleroot": "4d51591497f1d646070f9f9fdeb50dc338e2a8bb9a5cb721c55f452938165ff8",
"time": 1586364107,
"mediantime": 1586361287,
"nonce": 3963275925,
"bits": "171320bc",
"difficulty": 14715214060656.53,
"chainwork": "0000000000000000000000000000000000000000000e4eba1824303796d776922b",
"nTx": 2626,
"previousblockhash": "00000000000000000000068fb1ddc43ca83bc4bfb23444f7236992cfc565d40e08",
"nextblockhash": "000000000000000000010b3d94671593da669b25fecf7005de38dc2b2fa208dc7" }
```

Figure 5.2: Request and Result *getBlockHeader*

Beside the actual result the server provides the proof data. In case the client displays the result it was able to verify the data by using the proof data and performing the necessary proofs. By having a look at the debug information the proof data can be made visible.

```
...
"in3": {
    "proof": {
        "final": "0x00e00020d7cd60cab00255bb13bbb023fd5d85daaaf389720a1400000000...bb8784",
        "cbtx": "0x010000000001010000000000000000000000000000000000000000000000000...4ddd5c",
        "cbtxMerkleProof": "0xa22e7468d9bf239167ff6f97d066818b4a5278d29fc13dbcbd...4b2f3a"
    }
}
...
```

Figure 5.3: Debug Information *getBlockHeader*

The in3.proof-object contains the proof data:

- **final**: The finality headers of the requested block header, used to prove the finality of the requested block header. The number is set in the configuration of the client.
- **cbtx**: The coinbase transaction, used to extract the block number.
- **cbtxMerkleProof**: The Merkle proof, used to prove the existence and correctness of the coinbase transaction (and thereby the block number).

In case the data itself or the proof data was manipulated the client throws an error. To demonstrate that scenario a transport function can be used. This function allows to manipulate the response from the server before handed to the client.

```
1  const axios = require('axios')
2
3  IN3.setTransport(async (url, payload, timeout)=> {
4      const req =  JSON.parse(payload)
5      const result = await axios.post(url,req, { timeout, headers: { 'Content-Type': 'application/json' } })
6               .then(res => {
7                   if (res.status != 200) throw new Error("Invalid status")
8                   return res.data
9               })
10     // manipulate the response
11     if (req[0].method==='getblockheader') {
12         // remove last finality header
13         result[0].in3.proof.final = result[0].in3.proof.final.substring(0, result[0].in3.proof.final.length - 160)
14     }
15    return JSON.stringify(result)
16 })
```

Figure 5.4: Manipulate Response *getBlockHeader*

Line 13 in fig. 5.4 removes the last 160 characters of the in3.proof.final-field (i.e. the last 80 bytes). A Bitcoin block header has a size of 80 bytes, hence removing the last 80 bytes removes exactly one finality header. A limit of four attempts and eight finality headers are set in the configuration. After four attempts the request results in an error because the required number of eight finality headers could not be reached. Therefore, the finality could not be proven and the client was not able to verify the result. Additionally, the node is being blacklisted for sending an unverifiable response. The client is not going to send any further requests to this node. Figure 5.5 shows the error thrown by the client.

```
14:31:44 DEBUG execute.c:1026:in3_ctx_execute(): ctx_execute getblockheader ... attempt 4
14:31:44 TRACE context.c:168:ctx_set_error_intern(): Intermediate error -> Not enough finality blockheaders
14:31:44 DEBUG execute.c:352:blacklist_node(): Blacklisting node for unverifiable response: https://in3-v2.slock.it/btc/nd-2
Error: Not enough finality blockheaders
```

Figure 5.5: Manipulated Response Error *getBlockHeader*

Request Transaction

This request returns the transaction as a JSON-object for a given transaction hash.

```
async function sendRequest() {
    const tx = await in3.btc.getTransaction("f34d7883f22859bdbc3dc19fd278524a8b8166d0976fff679123bfd968742ea2")
    console.log(tx)
}

sendRequest().catch(console.error)

// Result:

{ "txid": "f34d7883f22859bdbc3dc19fd278524a8b8166d0976fff679123bfd968742ea2",
"hash": "f34d7883f22859bdbc3dc19fd278524a8b8166d0976fff679123bfd968742ea2",
"version": 2,
"size": 225,
"vsize": 225,
"weight": 900,
"locktime": 0,
"vin": [ { "txid": "49cc3d96b7d518a9a3875fecc5b7b4dc6dc57a57d7d192260afce24aaf26fc17",
    "vout": 1,
    "scriptSig": [...],
    "sequence": 4294967295 } ],
"vout": [ { "value": 0.01399427, "n": 0, "scriptPubKey": [...] },
    { "value": 0.0458431, "n": 1, "scriptPubKey": [...] } ],
"hex": "020000000117fc26af4ae2fc0a2692d1d7577ac56ddcb4b7c5ec5f87a3a918d5b7963dcc...5403c6375fd7f4288ac00000000",
"blockhash": "000000000000000000000140a7289f3aada855dfd23b0bb13bb5502b0ca60cdd7",
"confirmations": 27715,
"time": 1586364107,
"blocktime": 1586364107 }
```

Figure 5.6: Request and Result *getTransaction*

Beside the actual result the server provides proof data. In case the client displays the result it was able to verify the data by using the proof data and performing the necessary proofs. By having a look at the debug information the proof data can be made visible.

```
...
"in3": {
    "proof": {
        "block": "0x00e0ff3f080ed465c5cf926923f74434b2bfc43ba83cc4ddb18f0600000000...ca3aec",
        "final": "0x00e00020d7cd60cab00255bb13bbb023fd5d85daaaf389720a140000000000...bbd60f",
        "txIndex": 1,
        "merkleProof": "0x900278755603ba5d6ebe57135b9969fe7ad4590c60193308e0e97fe0...4b2f3a",
        "cbtx": "0x010000000001010000000000000000000000000000000000000000000000000000000000...4ddd5c",
        "cbtxMerkleProof": "0xa22e7468d9bf239167ff6f97d066818b4a5278d29fc13dbcbd59...4b2f3a"
    }
}
...
```

Figure 5.7: Debug Information *getTransaction*

The in3.proof-object contains the proof data:

- **block**: The block header of the block which contains the requested transaction. Required for the finality proof and the Merkle proofs.
- **final**: The finality headers of the requested block header, used to prove the finality of the requested block header. The number is set in the configuration of the client.
- **txIndex**: The index of the requested transaction, used for the Merkle proof of the requested transaction.
- **merkleProof**: The Merkle proof of the requested transaction, used to prove the existence and correctness of the requested transaction.
- **cbtx**: The coinbase transaction, used to extract the block number.
- **cbtxMerkleProof**: The Merkle proof for the coinbase transaction, used to prove the existence and correctness of the coinbase transaction.

In case the data itself or the proof data was manipulated the client throws an error. To demonstrate that scenario a transport function can be used. This function allows to manipulate the response from the server before handed to the client.

```
1  const axios = require('axios')
2
3  IN3.setTransport(async (url, payload, timeout)=> {
4      const req =  JSON.parse(payload)
5      const result = await axios.post(url,req, { timeout, headers: { 'Content-Type': 'application/json' } })
6               .then(res => {
7                   if (res.status != 200) throw new Error("Invalid status")
8                   return res.data
9               })
10     // manipulate the response
11     if (req[0].method==='getrawtransaction') {
12         // remove merkle proof
13         result[0].in3.proof.merkleProof = '0x'
14     }
15    return JSON.stringify(result)
16 })
```

Figure 5.8: Manipulate Response *getTransaction*

Line 13 in fig. 5.8 simply removes the Merkle proof data by setting it to "0x". After four attempts the requests results in an error because the client was not able to perform a Merkle proof for the requested transaction. Therefore, the existence and correctness of that transaction could not be proven. Additionally, the node is being blacklisted for sending an unverifiable response. The client is not going to send any further requests to this node.

```
1  17:45:22 DEBUG execute.c:1026:in3_ctx_execute(): ctx_execute getrawtransaction ... attempt 4
2  17:45:22 TRACE context.c:168:ctx_set_error_intern(): Intermediate error -> merkleProof failed!
3  17:45:22 DEBUG execute.c:352:blacklist_node(): Blacklisting node for unverifiable response: https://in3-v2.slock.it/btc/nd-2
4  Error: merkleProof failed!
```

Figure 5.9: Manipulated Response Error *getTransaction*

5.3 Further Demonstration

Better than any demonstration is to test it by yourself. Everyone can set up and run an IN3 client and even an IN3 server. Setting up a server is a little more tricky and takes some more effort. A full list of instructions can be found on readthedocs of IN3. For simplicity and immediate testing the connection to an IN3 server operated by Blockchains LLC is recommended (e.g. *https://in3-v2.slock.it/btc/nd-2*). [12]

There are two options for sending requests to the server:

- **IN3 Client**
  Choose one of the implementations of the IN3 client. For example, the WASM-client is running in browsers or node.js. It can easily be installed by using the node package manager: *npm install –save in3-wasm*. A detailed description, examples and more options of setting up an IN3 client can be found on IN3's readthedocs.

- **Postman**
  Using Postman allows sending simple HTTP-requests to a certain URL. It can be used to request data and to inspect the proof data besides the actual result. It does not perform any proofs and should be used as a data provider. The setup is fast and simple. This is the perfect options to explore the functionality of the IN3 server by inspecting the responses.

Important links:

- **IN3 readthedocs**
  `https://in3.readthedocs.io/en/develop/index.html`
- **IN3 Client GitHub**
  `https://github.com/blockchainsllc/in3-c`
- **IN3 Server GitHub**
  `https://github.com/blockchainsllc/in3-server`
- **Postman**
  `https://www.postman.com/`
- **Node Package Manager**
  `https://www.npmjs.com/`

6 Conclusion

This thesis has explained the fundamentals of Bitcoin, demonstrated the basics of the IN3 protocol and finally implemented the presented concept. The verification of Bitcoin in the IN3 protocol can be split into two parts: part one without and part two with on-chain conviction. At the time of writing, part one is completely implemented, documented and tested and is about to be released. Once released it will enable the users of IN3 to use and build applications that are using Bitcoin data in any manner.

The concept and a first implementation of part two was presented in this thesis. The implementation of the presented concept still needs to be completed to consider the verification of Bitcoin in the IN3 protocol as finished. This will most likely take a little more effort due to the complexity of this process and the high requirements of quality of code. The completion of the on-chain conviction and the optimization of the verification of blocks before BIP34 (as mentioned in 3.3.1) builds the future work of this thesis. The source code of the implemented Bitcoin modules in this work can be found on IN3's GitHub.

IN3 plans to not only support Ethereum and Bitcoin in the future. This work lays the foundation for many more blockchains to follow which will enable billions of IoT devices in the world to be connected to any blockchain they want.

Appendix A: Source Code of the On-chain Conviction

```
pragma solidity 0.6.6;

contract VerifyBlockHeader {

    function doubleHash(bytes memory header) pure public returns (bytes32){
        return sha256((abi.encodePacked(sha256((header)))));
    }

    function getBits(bytes memory header) pure public returns (bytes4 headerBits) {
        // use assembly
        // we load the provided blockheader
        // then we add 0x20 (32bytes) to get to the start of the blockheader
        // then we add 72 bytes (location of the headerBits)
        // and load it to the headerBits variable
        assembly {
            headerBits :=mload(
                add(
                    add(
                        header, 0x20
                    ), 72
                )
            )
        }
    }

    function swap(bytes32 input) public pure returns (bytes memory){
        bytes memory swapped = new bytes(input.length);

        for(uint i=0; i<input.length;i++){
            uint256 index = input.length-1-i;
            swapped[i]=input[index];
        }

        return swapped;
    }


    function verifyHeader(bytes memory header) public pure returns (bytes32 hash, bytes32 target, bool result) {
        bytes memory bytesHash = swap(doubleHash(header)); // get hash and swap order of bytes

        // convert to bytes32
        assembly {
            hash :=mload(
                add(
                    add(
                        bytesHash, 0x20
                    ), 0
                )
            )
        }

        bytes memory headerBits = swap(getBits(header)); // get bits out of header and swap order of bytes
        uint8 length = uint8(headerBits[28]); // bits are at [28][29][30][31]
                                              // first byte of the bits is the prefix (at [28])
                                              // convert hex to number

        // put the last 3 bytes of "headerBits" into "target"
        assembly {
            target :=mload(
                add(
                    add(
                        headerBits, 0x20
                    ), 29
                )
            )
        }

        target = target >> 8 * (32 - length); // shift the bytes to the correct spot
        result = (hash < target); // hash below target: true, otherwise false
    }

}
```

Listing A.1: Contract VerifyBlockHeader

Bibliography

[1] Käbisch, Tim. *Verification of Bitcoin in the Incubed protocol*. Conference proceedings of the Scientific Track of the Blockchain Autumn School 2020. Hochschule Mittweida. October 2020. Mittweida.

[2] Käbisch, Tim. *Technologien zu Verifizierung von Bitcoin auf der Ethereum Blockchain*. Internship Report. June 2020. Mittweida.

[3] *Internet of Things - active connections worldwide 2015-2025*. (2020, September 1). Statista. Online available: https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/ (visited on 10/28/2020).

[4] *More than 100 million Alexa devices have been sold*. (2019, January 4). TechCrunch. Online available: https://techcrunch.com/2019/01/04/more-than-100-million-alexa-devices-have-been-sold/ (visited on 10/28/2020).

[5] Panetta, K. *5 Trends Emerge in the Gartner Hype Cycle for Emerging Technologies, 2018*. (2018, August 18). Smarter with Gartner. Online available: https://www.gartner.com/smarterwithgartner/5-trends-emerge-in-gartner-hype-cycle-for-emerging-technologies-2018/ (visited on 10/28/2020).

[6] *Blockchain Charts*. Blockchain.com. Online available: https://www.blockchain.com/charts/ (visited on 10/28/2020).

[7] Walker, Greg. *Mining - How does mining work in Bitcoin*. (2020, July 21). learnmeabitcoin. Online available: https://learnmeabitcoin.com/technical/mining (visited on 10/26/2020).

[8] Antonopoulos, A. M. (2017). *Mastering Bitcoin: Programming the Open Blockchain (2nd ed.)*. O'Reilly Media. ISBN: 878-1-491-95438-6.

[9] Antonopoulos, A. M., & D., G. W. P. (2018). *Mastering Ethereum: Building Smart Contracts and DApps (1st ed.)*. O'Reilly Media. ISBN: 978-1-491-97194-9.

[10] Walker, Greg. *Difficulty - What is Difficulty in Bitcoin*. (2020, March 28). learnmeabitcoin. Online available: https://learnmeabitcoin.com/technical/difficulty (visited on 03/18/2020).

[11] Walker, Greg. *Bits*. (2020, July 21). learnmeabitcoin. Online available: https://learnmeabitcoin.com/technical/bits (visited on 03/18/2020).

[12] Kux, Steffen & Jentzsch, Simon. *IN3's documentation*. (2019). Read the Docs. Online available: https://in3.readthedocs.io/en/develop/intro.html.

[13] *Products: IN3 - A closer look*. (2020). Blockchains. Online available: https://www.blockchains.com/our-products/incubed/ (visited on 10/27/2020).

[14] Kux, Steffen & Jentzsch, Simon. *IN3's documentation - Incentives*. (2019). Read the Docs. Online available: https://in3.readthedocs.io/en/latest/intro.html#validated-requests (visited on 10/27/2020).

[15] Nakamoto, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System*. (2008, October 31). Online available: https://bitcoin.org/bitcoin.pdf (visited on 08/25/2020).

[16] *IN3 Client in C*. GitHub. Online available: https://github.com/blockchainsllc/in3-c (visited on 10/16/2020).

[17] *Bitcoin Improvement Proposal (BIPs)*. (2019, March 25). BitcoinWiki. Online available: https://en.bitcoinwiki.org/wiki/Bitcoin_Improvement_Proposals (visited on 10/05/2020).

[18] *Bitcoin/Bips/Bip-0034*. (2016, November 30). GitHub. Online available: https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki (visited on 10/05/2020).

[19] *Antminer S9 SE-16TH/s*. Bitmain. Online available: https://shop.bitmain.com/product/detail?pid=00020200306153650096S2W5mY1i0661 (visited on 10/05/2020).

[20] Ittner, Andreas. *Distributed Systems (Lecture)*. Hochschule Mittweida. (Winter term 2019/20).

[21] Prahalad, Belavadi. *Merkle proofs Explained*. (2018, January 7). Medium. Online available: https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5 (visited on 08/20/2020).

[22] *Transaction Info*. Blockchair. Online available: https://blockchair.com/bitcoin/transaction/02d8cdb103f50532e2f18d9d1f85c016468ee0294908d387e38f80b99410d893 (visited on 08/21/2020).

[23] *Blocknumber not part of the header*. (2020, April 3). Bitcoin Forum. Online available: https://bitcointalk.org/index.php?topic=5237590 (visited on 04/03/2020).

[24] *Preimage attack*. (2020, July 26). Wikipedia. Online available: https://en.wikipedia.org/wiki/Preimage_attack (visited on 10/06/2020).

[25] *Hash power Marketplace*. NiceHash. Online available: https://www.nicehash.com/marketplace (visited on 09/04/2020).

[26] Walker, Greg. *Longest Chain*. (2020, July 21). learnmeabitcoin. Online available: https://learnmeabitcoin.com/technical/longest-chain (visited on 10/08/2020).

[27] Rouse, M. *Remote Procedure Call (RPC)*. (2020, May 4). SearchAppArchitecture. Online available: https://searchapparchitecture.techtarget.com/definition/Remote-Procedure-Call-RPC (visited on 10/16/2020).

[28] *Bitcoin Core 0.18.0 RPC*. (2020). Bitcoin Core. Onnline available: https://bitcoincore.org/en/doc/0.18.0/ (visited on 07/22/2020).

[29] *Cache*. Merriam-Webster. Online available: https://www.merriam-webster.com/dictionary/cache (visited on 10/21/2020).

[30] *Code Coverage IN3 Server*. Internal source. (data retrieved on 10/20/2020).

[31] Oehlert, Peter. *Violating Assumptions with Fuzzing*. IEEE Security & Privacy. (2005 March/April). Online available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1423963 (visited on 11/01/2020)

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 02. November 2020