VERIFICATION OF BITCOIN IN THE INCUBED PROTOCOL

Tim Käbisch
Slock.it GmbH, Markt 16, D-09648 Mittweida

Abstract

To enable smart devices of the internet of things to be connected to a blockchain, a blockchain client needs to run on this hardware. With the Trustless Incentivized Remote Node Network, in short Incubed, it will be possible to establish a decentralized and secure network of remote nodes, which enables trustworthy and fast access to a blockchain for a large number of low-performance IoT devices. Currently, Incubed supports the verification of Ethereum data. To serve a wider audience and more applications this paper proposes the verification of Bitcoin data as well, which can be achieved due to the modularity of Incubed. This paper describes the proof data that is necessary for a client to prove the correctness of a node's response and the process to verify the response by using this proof data as well. A proof-object which contains the proof data will be part of every response in addition to the actual result. We design, implement and evaluate Bitcoin verification for Incubed. Creation of the proof data for supported methods (on the server-side) and the verification process using this proof data (on the client-side) has been demonstrated. This enables the verification of Bitcoin in Incubed.

1. Introduction

"The blockchain data structure is an ordered, back-linked list of blocks of transactions. Each block within the blockchain is identified by a hash, generated using a cryptographic hash algorithm on the header of the block. Each block references a previous block, known as the *parent* block, through the "previous block hash" field in the block header. The sequence of hashes linking each block to its parent creates a chain going back all the way to the first block ever created, known as the *genesis block*." [1]

At the moment the most famous use case of the blockchain technology are cryptocurrencies. Besides Bitcoin and Ethereum there are around 6,500 [2] more at the time of writing. Many more use cases are about to follow and are currently being developed.

To enable the interaction with a blockchain a device needs to install and run a blockchain client. While current notebooks and desktop computers have enough computational power, storage space and bandwidth to run a full node, smaller devices like smartphones or tablets with less powerful hardware or restricted internet connection are capable of running a light node (also known as SPV-client for Bitcoin). However, many IoT devices are severely constrained in terms of computational power and internet connection that even a light node is too "big" to run on such devices. Connecting an IoT device to a remote node still enables the connection to a blockchain. But, by using remote nodes a big advantage of a decentralized network is being undermined: not being forced to trust a single player. The risk of malfunction or an attack is very high since there is a single point of failure. Therefore, we need a blockchain client which is small enough to run on an IoT device and which can act independently in a network of players, hence not being forced to trust a single node. [3]

"The Trustless Incentivized Remote Node Network, in short Incubed, 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." [4]

2. Incubed

Incubed solves the following problems which are preventing an IoT device to run a light node: [5]

1. **Insufficient computing power and storage space:** Incubed takes up very little storage space (200 kB for the implementation in C).
2. **Insufficient power supply:** Incubed does not require a continuous power supply and therefore can be switched off after each usage.
3. **No continuous connection to the internet:** Incubed does not need a continuous internet connection, but only builds it if required – this is only possible because Incubed is a non-synchronizing client.

The Incubed network consists of the following components: [6]

1. **Incubed Registry:** This is a smart contract on the Ethereum blockchain. Nodes that want to participate in the network must register and store a security deposit.
2. **Incubed Node:** Full nodes of a blockchain which provide information and act as a validator.
3. **Incubed Client:** Clients that request information from Incubed nodes and can be installed on IoT devices, among others.
4. **Watchdogs:** Autonomous authorities (bots) who are responsible for the detection and punishment of fraudulent nodes.

Figure 1 shows the flow of an RPC request in the Incubed network. The client sends a request to Node B and requests signatures from Nodes A and C on the provided answer by B. Node B sends its (unsigned) response to the requested signature nodes. They check the response and answer B with their signature if the answer from B is correct. After receiving the signatures from the required nodes B will send a response to the client including the actual result and the signatures. The client assumes that the

majority of the nodes act honest and therefore considers the result as correct once he received and checked the corresponding signatures. In case B tries to send a fake answer to the client honest nodes would not sign this reply and in addition convict node B in the registry. Node B would then be removed from the registry and would lose his security deposit.

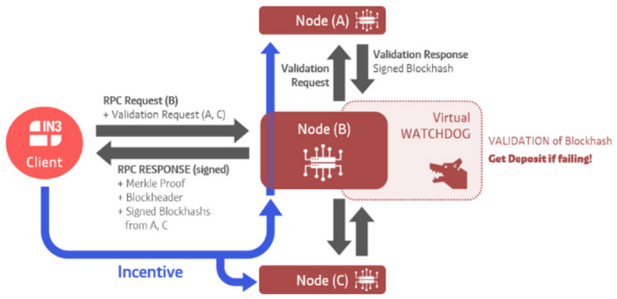

Fig. 1: Flow of an RPC request in the Incubed network

"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. […] 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)." [7]

Lets have a look at a real world example of the usage of Incubed: a **smart bike lock**. The rental of an e-bike will be managed by a smart contract deployed on Ethereum. The lock is powered by the battery of the e-bike. It is equipped with a microchip 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 Incubed 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. [8]

Currently Incubed supports the verification of Ethereum data. Supporting Bitcoin, which is the largest cryptocurrency with a market cap of $210.89B and around 320,000 transactions per day [9,10], will serve a wider audience and enables many use cases for the users of Incubed. The most important use case is the verification of payments on the Bitcoin chain. Incubed clients will be able to prove the existence and correctness of a transaction (a use case here would be online shop payments). There are many more applications based on Bitcoin, for example a "proof of existence" (storing the hash of a document on the Bitcoin chain - github.com/proofofexistence).

3. Fundamentals

For the verification of Bitcoin we make use of the Simplified Payment Verification (SPV) proposed in the Bitcoin 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." [11]

In contrast to SPV-clients an Incubed client does not keep a copy of all block headers, instead the client is stateless and only requests required block headers. We are following a simple process: A client requests certain data, the server sends a response with proof data in addition to the actual result, the client verifies the result by using the proof data. We rely on the fact that it is extremely expensive for an attacker to deliver a wrong block (wrong data) which still has following blocks referring the wrong block (i.e. delivering a chain of fake-blocks). This approach does not really work for very old blocks. Beside the very low difficulty at this time, the miner has many years of time to pre-mine a wrong chain of blocks. Therefore, we are setting some hard-coded checkpoints of hashes of bygone blocks. Proving the correctness of old blocks can be achieved by checking the linking from the requested block to a certain checkpoint (the server needs to provide the corresponding data). The only way for an attacker to fool the client would be by finding a hash collision.

3.1 Mining in Bitcoin

The process of trying to add a new block of transactions to the Bitcoin blockchain is called "mining". Miners are competing in a network-wide competition, each trying to find a new block faster than anyone else. The first miner who finds a block broadcasts it across the network and other miners are adding it to their blockchain after verifying the block. Miners restart the mining-process after a new block was 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.

But how can miners FIND a block? They start by filling a candidate block with transactions from their memory pool. Next they construct a block header for this block, which is a summary of all the data in the block including a reference to a block that is already part of the blockchain (known as the parent hash). Now the actual mining happens: miners put the block header through the *SHA256* hash function and hope that the resulting hash is below the current target. If this is not the case, miners keep trying by incrementing a number in the block header resulting in a completely different hash. This process is referred to as **proof-of-work.** [12]

3.2 Finality in Bitcoin

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. [13]

3.3 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**.

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 timestamps of the first and last block). The difficulty increases when the blocks are mined faster than expected and vice versa. Although the computing power increased heavily since the introduction of Bitcoin in 2009 the average block time is still 10 minutes due to this mechanism.

What is the difference between the difficulty and the target? The difficulty is a big number used for the adjustment process. The target is used for the mining process and for the verification of a block hash. The hash of a block has to be smaller than the target to be accepted across the network (as mentioned in 3.1). The target can be calculated using the difficulty and the constant value *targetmax*:

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

$$targetmax = 0x00000000FFFF00000000000000000000000000000000000000000000000000000000$$

4. Risk Calculation

The following calculation outlines the security (in terms of $) when the client is requesting one of the newer blocks and 6 finality headers. This results in a total of 7 fake-blocks that an attacker has to calculate to fool the client. The calculation is based on assumptions and averages.

Assume that the attacker has 10% of the total mining power. This would mean he needs around 100 minutes to mine 1 block (average block time of Bitcoin is 10 minutes) and around 700 minutes to mine 7 blocks. While mining fake-blocks, the attacker loses his chance of earning block rewards. Assuming that we would have been able to mine 7 blocks, with a current block reward of 6.25 BTC and $11,400 per Bitcoin at the time of writing [14]:

$$7 * 6.25\ BTC = 43.75\ BTC$$

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

Furthermore, the attacker needs to achieve 10% of the mining power. With a current total hash rate of 120 EH/s, this would mean 12 EH/s. There are two options: buying the hardware or renting the mining power from others. [15]

A new Antminer S9 with 14 TH/s can be bought for $3,000 [16]. This would mean an attacker has to pay $2,568,000,000 to buy so many of these miners to reach 12 EH/s. The costs for electricity, storage room and cooling still needs to be added.

Hashing power can also be rented online. Obviously nobody is offering to lend 12 EH/s of hashing power – but for this calculation we assume that an attacker is still able to rent this amount of hashing power. The website nicehash.com is offering 1 PH/s for 0.0098 BTC (for 24 hours). [17]

$$1\frac{PH}{s} = 0.0098\ BTC \rightarrow 12\frac{EH}{s} = 117.6\ BTC$$

Assuming it is possible to rent it for 700 minutes only (which would be 48.6% of one day).

$$117.6\ BTC * 0.486 = 57.15\ BTC$$

$$57.15\ BTC * \frac{\$11{,}400}{1\ BTC} = \mathbf{\$651{,}510}$$

Therefore, 6 finality headers provide a security of estimated **$1,150,260** in total. Figure 2 uses the assumptions and numbers from the calculation shown above to outline the costs to mine *n* fake-blocks.

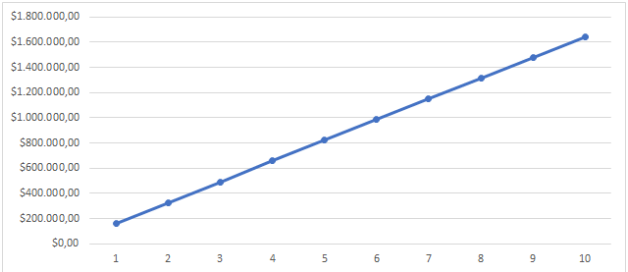

Fig. 2: Cost to mine n fake-blocks

5. Proofs

A subset of the following proofs needs to be performed to prove the correctness of certain data. The subset depends on the data itself.

5.1 Target Proof

Bitcoin uses the target for the mining process where miners are hashing the block data over and over again to find a hash that is smaller than the target (while changing the data a little each try to generate a different hash). Miners across the network can verify newly published blocks by checking the block hash against the target. The same applies for clients. Having a verified target on the client-side is important to verify the proof of work and therefore the data itself (assuming that the data is correct when someone put a lot of work into it). Since the target is part of a block header (*bits*-field) we can verify the target by verifying the block header. This is a dilemma since we want to verify the target by verifying the block header, but we need a verified target to verify the block header (explained in 5.2).

There are two options to verify a target, whereby only option one is implemented at time of writing - option two has not been implemented yet and is still being discussed.

5.1.1 Verification using finality headers

The client maintains a cache with the number of a difficulty adjustment period (dap) and the corresponding target (which stays the same for the duration of one period). This cache will be filled with default values at the time of the release of the Bitcoin implementation. If a target is not yet part of the cache it needs to be verified first and added to the cache afterwards.

We completely rely on the finality of a block. We can verify the target of a block (and therefore for a whole period) by requesting a block header (*getblockheader*) and *n*-amount of finality headers. If we are able to prove the finality using the finality proof we can consider the target as verified. The client sets a limit in his configuration regarding the maximum change of the target from a verified one to the one he wants to verify. The client will not trust the changes of the target when they are too big (i.e. greater than the limit). For such cases we implemented a special *proofTarget*-method to verify big changes of the target in smaller steps (explained in 6.7)

5.1.2 Verification using signatures

This approach uses the design of Incubed: verify a result by requesting signatures from other nodes. At time of writing this approach is still in development and discussion for Bitcoin.

Since the target is part of the block header we just have to be very sure that the block header is correct - which leads us to a correct target. The client fetches the node list and chooses n nodes which will provide a signature. Afterwards it sends a *getblockheader*-request (also containing the addresses of the selected nodes) to a random provider node. This node asks the signatures nodes to sign his result (the block header). The response will include the block header itself and all the signatures as well. The client can verify all signatures by using the node list and therefore verifying the actual result (a verified block header and therefore a verified target). The incentivization for the nodes to act honestly is their deposit which they will lose in case they act maliciously. Have a look at Fig. 1 for a better explanation of this process.

5.2 Block Proof

Verifying a Bitcoin block is quite easy when you already have a verified block hash. We take the first 80 bytes of the block data (which is the block header) and hash it with *SHA256* twice. Since Bitcoin stores the hashes in little endian we have to reverse the order of the bytes. In order to check the proof of work in the block header we compare the target with the hash. We accept the proof of work when the block hash is smaller than the target.

5.3 Finality Proof

Necessary data to perform this proof:

- Block header (block *X*)
- Finality block header (block *X+1*, …, *X+n)*

The finality for block *X* can be proven as follows: The proof data contains the block header of block *X* as well as *n* following block headers as finality headers. In Bitcoin every block header includes a *parentHash*-field which contains the block hash of its predecessor. By checking this linking the finality can be proven for block *X*. Meaning the block hash of block *X* is the *parentHash* of block *X+1*, the hash of block *X+1* is the *parentHash* of block *X+2*, and so on. If this linking is correct until block *X+n* (i.e. the last finality header) then block *X* can be considered as final. As mentioned earlier Bitcoin uses a probabilistic finality, meaning a higher *n* increases the probability of being actual final.

5.4 Transaction Proof

Necessary data to perform this proof:

- Block header
- Transaction
- Merkle proof
- Index (of this transaction)

All transaction 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 us to verify and prove the existence and correctness of a certain transaction.

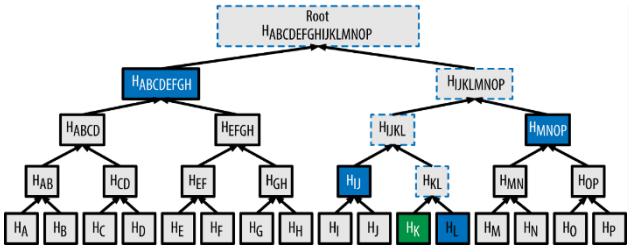

Fig. 3: Merkle Tree

In order to verify the existence and correctness of transaction [K] we use *SHA256* 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 Fig. 2. 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 we've hence proven the existence and correctness of transaction [K]. 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. [18, 19]

5.5 Block Number Proof

Necessary data to perform this proof:

• Block header

• Coinbase transaction (first transaction of the block)

• Merkle proof (for the coinbase transaction)

In comparison to Ethereum there is no *block number* in a Bitcoin block header. Bitcoin uses the height of a block, which is the number of predecessors. The genesis block is at height 0 since there are no predecessors (the block with 100 predecessors is at height 100). Therefore, you need to know the complete Bitcoin blockchain to verify the height of a block (by counting the links back to the genesis block). Hence, actors that do not store the complete chain (like an Incubed client) are not able to verify the height of a block (i.e. the number). To change that Gavin Andresen proposed a change to the Bitcoin protocol in 2012.

Bitcoin Improvement Proposal 34 (BIP-34) 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 all blocks must include the block height in their coinbase transaction. [20]

For all blocks after block number 227,835 the block number can be proven as follows:

1. Extract block number out of the coinbase transaction:

Coinbase transaction of block #624692 [21]

```
03348809041f4e8b5e7669702f7777772e6f6b657
82e636f6d2ffabe6d6db388905769d4e3720b1e59
081407ea75173ba3ed6137d32308591495198155c
e020000004204cb9a2a31601215b2ffbeaf1c4e00
```

Decode:

a) **03**: first byte signals the length of the block number (push the following 3 bytes)

b) **348809**: the block number in big endian format (convert to little endian)

c) **098834**: the block number in little endian format (convert 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:

To trust the extracted block number it is necessary to verify the existence and correctness of the coinbase transaction. This can be done by performing a *merkle proof* (explained in 5.4) using the provided block header and the merkle proof data. The size of the block number proof is **764 bytes** on average. [22]

6. Implementation

The implementation is split into two parts: the typescript-based server and the client based on C. The implementation in the server was done by Tim Käbisch, while Simon Jentzsch, Vice President of Blockchain Development at Blockchains LLC, did the implementation in the client. [23] An Incubed server acts as a kind of proxy server. In comparison to a standard Bitcoin full node an Incubed server is going to provide the corresponding proof data to verify the result on the client-side.

There are two options for an Incubed server to access Bitcoin data: Running a Bitcoin fullnode on the same machine as the Incubed server or connecting to a remote (trusted) Bitcoin full node. The operator of the Incubed 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 Incubed server on the same machine (or in the same trusted network) is highly recommended.

We found a way to prove the results of the standard Bitcoin RPC request mentioned in 6.2 - 6.6 by adding proof data and performing the corresponding proofs on the client-side. A proof object contains a subset of the following properties:

• *final* the finality headers, which are hex coded bytes of the following headers (80 bytes each) concatenated, the number depends on the requested

finality (*finality*-property in the *in3*-section of the request)

• *cbtx* serialized coinbase transaction of the block (this is needed to get the verified block number)

• *cbtxMerkleProof* merkle proof of the coinbase transaction, proving the correctness of the cbtx

• *block* a hex string with 80 bytes representing the block header

• *txIndex* index of the transaction (*txIndex=0* for coinbase transaction, necessary to create/verify the merkle proof)

• *merkleProof* merkle proof of the requested transaction, proving the correctness of the transaction

6.1 Cache

Server. The server is using a simple map with the block hash and the block number pointing to the same cache-object. A cache-object consists of the block height, block hash, block header, transactions ids and the coinbase transaction.

Client. The client is using a map with the number of a difficulty adjustment period (dap) pointing at the (verified) target of this dap. At the time of release we will fill this cache with default values for bygone daps. Every time the client can't find a needed target in his cache he is going to verify it by using the target proof (mentioned in 5.1) and stores it in the cache afterwards. Additionally, the client can use the proofTarget-method (mentioned in 6.7) to verify unusually large changes of the target.

6.2 getblockheader

Server. Returns data of block header for given block hash (returned level of details depends on *verbosity*). The server adds finality headers, the coinbase transaction and the merkle proof for the coinbase transaction to the actual result.

Client. Proves the result by performing a finality proof and block number proof.

6.3 getblock

Server. Returns data of block for given block hash (returned level of details depends on *verbosity*). The server adds finality headers, the coinbase transaction and the merkle proof for the coinbase transaction to the actual result.

Client. Uses the proof data to verify the result by performing a finality proof and a block number proof.

6.4 getrawtransaction

Server. Returns the raw transaction data (returned level of details depends on *verbosity*). The server adds the block header, finality headers, transaction index, merkle proof for the requested transaction, coinbase transaction and merkle proof for the coinbase transaction.

Client. The block header and the finality headers are used to perform a finality proof. By doing a merkle proof using the transaction index and the merkle proof for the requested transaction the correctness can be proven. Furthermore, the client is going to perform a block number proof using the coinbase transaction and its merkle proof.

6.5 getblockcount/getbestblockhash

The functionality of these two methods is the same, only the return value is a little different – the number of blocks in the longest chain for *getblockcount* and the hash of the best block in the longest chain for *getbestblockhash*. Since the server can not prove the finality of the latest block (obviously the finality headers are not existing yet) we consider the *current block count MINUS amount of finality* (set in the request) as the latest block.

Server. Returns "latest" block number/block hash of the longest chain. The server adds the block header, finality headers, coinbase transaction and the merkle proof for the coinbase transaction to the actual result.

Client. Proves the result by performing a finality proof and block number proof.

6.6 getdifficulty

Server. Returns the proof-of-work difficulty as a multiple of the minimum difficulty. Depending on the parameter the server will return the difficulty of a certain block number or the difficulty of the "latest" block (latest in the same sense as mentioned in 6.5).

Client. Uses the proof data to verify the result by performing a finality proof and a block number proof. Since the difficulty is not part of the block it can be checked by transforming it into a target (as mentioned in 3.3).

6.7 proofTarget

Whenever the client is not able to trust the changes of the target (which is the case if a block can not be found in the verified target cache *and* the value of the target changed more than the client's limit *max_diff*) he will call this method. It will return additional proof data to verify the changes of the target on the client-side. The server will provide a path of daps with corresponding proof data. By performing a finality proof and a block number proof for each dap of the path the client is able to verify the changes of the target step by step instead of having to trust a big change of the target (see Fig. 4).

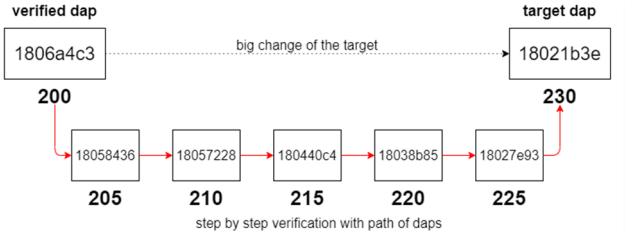

Fig. 4: Visualization of the *proofTarget*-method

7. Example

This example shows a *getblockheader*-request from the client and the corresponding response from the server including the proof data.

Request:

```
{
  "jsonrpc": "2.0",
  "id":1,
  "method": "getblockheader",
  "params":
["0000000000000000000103b2395f6cd94221b10d02
eb9be5850303c0534307220", false],
  "in3": {
    "finality": 8,
    "verification": "proof"
  }
}
```


Response:


```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result":
"00e00020497f4c193dbb347c2ecfcf6169e64c7478
778340a0cb13000000000000000000cb59b377938a5
32525d52b8404390f17c35a50beb9d57292f3658cea
34a386d7e4888d5e413b14174f045476",
  "in3": {
    "proof": {
      "final": "0x00e0ff27207230...276470",
      "cbtx": "0x010000000001010...9da2fc",
      "cbtxMerkleProof": "0x6a807...52e688"
    },
  "lastNodeList": 3290882,
  "execTime": 1459,
  "rpcTime": 0,
  "rpcCount": 0,
  "currentBlock": 3290942,
  "version": "2.1.0"
  }
}
```


The client can verify the result by itself by performing the corresponding proofs. The finality headers from the final-field will be used to perform a finality proof. To verify the block number the client is going to perform a block number proof using the coinbase transaction (cbtx-field) and the merkle proof for the coinbase transaction (cbtxMerkleProof-field).

8. Conclusion

This paper has shown the basics of the Incubed protocol, fundamentals of Bitcoin, necessary proofs and the actual implementation of the verification of Bitcoin in the Incubed protocol. At the time of writing we are about to release version 1 of the Bitcoin implementation. This enables the verification of Bitcoin in the Incubed protocol. Version 2 is still in the discussion and development phase. Once implemented, it will enable the on-chain conviction for malicious Bitcoin nodes. This also enables the verification of a balance of a Bitcoin address (since the balance is the sum of all UTXOs and this can be checked by every node – nodes that deliver a wrong result will surely be convicted).

The documentation can be found on Incubed's readthedocs and as soon as version 1 of the Bitcoin implementation is released it will be available on GitHub.

Documentation: https://in3.readthedocs.org/

Client: https://github.com/blockchainsllc/in3-c

Server: https://github.com/blockchainsllc/in3-server

9. References

[1] Andreas M. Antonopoulos, Mastering Bitcoin (2017), 211

[2] CoinMarketCap (19.08.2020) https://coinmarketcap.com/de/all/views/all/

[3] Incubed Readthedocs (18.08.2020) https://in3.readthedocs.io/en/develop/intro.html

[4] Incubed GitHub (18.08.2020) https://github.com/blockchainsllc/in3

[5] Incubed Readthedocs (19.08.2020) https://in3.readthedocs.io/en/develop/intro.html#low-performance-hardware

[6] Incubed Readthedocs (19.08.2020) https://in3.readthedocs.io/en/latest/intro.html#architecture

[7] Incubed Readthedocs, Payment and Incentives (19.08.2020) https://in3.readthedocs.io/en/latest/intro.html#validated-requests

[8] Incubed Readthedocs, Smart Lock (20.08.2020) https://in3.readthedocs.io/en/latest/intro.html#smart-bike-lock

[9] Bitcoin Statistics (27.08.2020) https://www.coindesk.com/price/bitcoin

[10] Bitcoin Statistics (27.08.2020) https://www.blockchain.com/charts

[11] Bitcoin Paper (25.08.2020) https://bitcoin.org/bitcoin.pdf

[12] Learnmeabitcoin, Mining https://learnmeabitcoin.com/technical/mining

[13] Medium, Finality in Blockchain Consensus (19.08.2020) https://medium.com/mechanism-labs/finality-in-blockchain-consensus-d1f83c120a9a

[14] Coindesk (28.08.2020) https://www.coindesk.com/price/bitcoin

[15] Bitcoin Hash Rate (28.08.2020) https://charts.bitcoin.com/btc/chart/hash-rate

[16] Antminer S9 (28.08.2020) https://www.buybitcoinworldwide.com/mining/hardware/

[17] Nicehash (28.08.2020) https://www.nicehash.com/marketplace

[18] Medium, Merkle Proof (20.08.2020) https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5

[19] Wikipedia, Merkle Tree (20.08.2020) https://en.wikipedia.org/wiki/Merkle_tree

[20] GitHub, BIP-0034 (21.08.2020) https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki

[21] Blockchair, Coinbase Transaction (21.08.2020) https://blockchair.com/bitcoin/transaction/02d8cdb103f50532e2f18d9d1f85c016468ee0294908d387e38f80b99410d893

[22] Incubed Readthedocs, Block Number Proof Size https://in3.readthedocs.io/en/latest/bitcoin.html#block-number-proof

[23] Blockchains LLC (28.08.2020) https://www.blockchains.com/