Enabling NIPoPoW Applications on Bitcoin Cash

Kostis Karantias
ID: *2454*

Diploma Thesis

Supervisor: Prof. Leonidas Palios

Ioannina, February 2019

Department of Computer
Science & Engineering
University of Ioannina

Abstract

Cryptocurrencies have seen a meteoric rise in popularity in the last 10 years since the invention of Bitcoin. It is typical for cryptocurrencies to require every participant to keep a record of all the history. As usage skyrocketed this solution obviously did not scale. The Bitcoin paper includes a solution called SPV, which reduces the amount of information a participant needs to know, however the information is still linear in chain size. NIPoPoWs (Non-Interactive Proofs of Proof-of-Work) are a new advancement, providing an improvement to logarithmic in chain size instead of linear, however the model assumed by NIPoPoWs is not directly compatible with any of the popular cryptocurrencies.

In this work we explore the implementation of NIPoPoWs on Bitcoin-based cryptocurrencies. We study how a chain can be augmented with a crucial data structure called the interlink via a User-Activated Velvet Fork. We design software called the *interlinker*, which creates such a fork on a given chain. We provide two implementations of an interlinker, one which requires a full node and one which requires a lite node, both released as open-source software. We deploy our solutions on the Bitcoin Cash testnet, and produce reliability metrics for our deployment. Furthermore, we introduce a new approach for encoding interlink commitments in blocks, called *interlink compression*, which reduces proof sizes even further.

Finally, we provide a working implementation of a *prover*, also released as open-source. The prover enables users to generate suffix and infix proofs on any Bitcoin-based chain which has been velvet forked using our method, including the Bitcoin Cash testnet, where our fork has been deployed. The proofs generated can then be used for super-lite clients, cross chain transactions, and more.

Acknowledgements

Two people were key in getting me interested in algorithms, programming and computer science when I was still in high-school: Aleksis Brezas and Dionysis Zindros. I am indebted to both of them for the time they spent teaching me back then, and I couldn't have asked for more sincere and capable people to surround myself with. Our friendship has helped me grow both as a scientist and as a person. I also wish to thank Dimitris Glezos for believing in me when I was younger and indirectly introducing me to these two.

The topic of this thesis was suggested by Dionysis, who is also one of the authors of the NIPoPoWs paper that this thesis builds upon. He graciously spent time with me navigating the complex proofs of that paper, which gave me a much deeper understanding of the topic. He also made sure that I felt welcome in Decrypto, the research group he is in, which I joined and subsequently completed this work in.

During my studies I worked for 4 months as an intern for Bloomberg in London, where I was incredibly lucky to have Hang Xu as my mentor, who really helped me solidify my confidence as a software engineer. I was also lucky to have great co-workers who too made my time there really special.

I also want to thank my friends who have been by my side all these years. My friends who I've worked with and my friends who always had the patience to listen to my rantings and bad jokes. You all know who you are but I will name a few: Marios Balamatsias, Giannis Goulioumis, Kostas Krommydas, Christos Porios, Themis Papameletiou and Dimitris Lamprinos.

I am grateful to have had some great professors to teach me during my time as an undergraduate. I am even more grateful to have had professors who sincerely were there for me during hard times and always had time to talk. I will mention two of them: my supervisor, Leonidas Palios and Christos Nomikos. I feel incredibly fortunate to know them and to have been taught by both of them.

Finally, I thank my family for raising me and for continuously supporting me during my studies. It goes without saying that I would not be where I am today without them.

Thank you all. It has been an incredible journey and it would not be the same without any one of you.

Contents

Chapter 1

Introduction

1.1 Motivation

Cryptocurrencies are digital assets that utilize cryptography in order to allow value transfer, without the need of a central party or trusted authority. The technology originally appeared in 2008 in a paper by Satoshi Nakamoto [26] as Bitcoin, along with a reference implementation in C++. It didn't take long until a community of enthusiasts and cryptographers embraced the technology and started studying it and using it extensively. New cryptocurrencies based on Bitcoin's ideas and codebase started popping up, among them most notably Litecoin and Dogecoin.

This work clearly was a huge inspiration. In 2014 Ethereum [5, 32] appeared, which aimed to do much more than just value transfers: it built on Nakamoto's ideas in order to build a world computer. Programs called *smart contracts* could be stored and run in a decentralized manner. Such smart contracts gave us the ability to write immutable contracts, where "code is law."

Few years later and the landscape is completely changed. More and more cryptocurrencies are created every single day. Public interest and prices have skyrocketed. There is lots of optimism about the future decentralized technologies such as Bitcoin can bring, mainly a democratization of money, usually called "banking the unbanked."

However, being more widespread and popular surfaced some problems the most important of which is scalability. Technically, each cryptocurrency has a *blockchain*, which is literally a chain of *blocks* linked together like a linked list. These blocks contain *transactions*. Every transaction needs to be recorded and stored in a block, and everyone has to know about it. As a result, the Bitcoin blockchain is 185GB at the time of writing. The Ethereum blockchain comes up to 667GB. Typically, for someone to participate on the network, and do actions like send transactions they have to download the whole blockchain. However at such rates it is very time-consuming and resource-intensive or even impossible for someone to download a chain. So-called *lite nodes* that don't need to download the whole chain do exist, but at best they need information linear in the size of the chain, so they're a constant-factor improvement.

New cryptocurrencies with interesting features pop up all the time. There's long been an interest in implementing sidechains [2], as a way to interoperate between two blockchains. One should be able to trustlessly transfer his Bitcoin to another chain and use it there (a *one-way peg*), and transfer it back if he so desires (a *two-way peg*). One-way pegs have been implemented by having a smart contract on a cryptocurrency like Ethereum which knows the full state of the other blockchain, much like a full node. However storage comes at a huge cost, which makes the hundreds of GBs of most blockchains infeasible to maintain.

There's also the issue of user experience: even using a lite node at this point is too slow. Lite nodes use a protocol called *SPV* which we'll look into detail shortly. SPV works by relaying only the block headers (which are of constant size) to the lite nodes and specifically any transactions they might suspect will interest them. Even so, the Dogecoin blockchain headers come up to 188MB. On an Android phone using the Dogecoin app, it takes about an hour before the app can be used, and this is assuming good network conditions. Trying to use such clients on a data plan could potentially make them completely unusable.

NIPoPoWs [18] are a recent blockchain primitive allowing proofs on the blockchain of only

logarithmic size instead of linear. These succinct proofs turn out to be key in solving problems like the above. One-way pegged contracts don't need to know about the whole blockchain, and lite nodes don't need to know all block headers to have the same security assurances.

1.2 Related Work

The first solution for short proofs came directly from the original Bitcoin paper [26] in the form of SPV. The main idea of SPV, Merkle Tree Proofs, which we will be studying in depth shortly, is very useful for transaction inclusion proofs with NIPoPoWs as well.

Although NIPoPoWs are a very recent development, implementations have already been developed. Most notably, a verifier smart contract has been developed by Christoglou [7], which validates and compares NIPoPoW proofs to find the best, and can be used as a basis for cross-chain transactions. This implementation however assumes proofs on a blockchain which contains the interlink on its headers, which is something untrue for most popular cryptocurrencies.

There are however cryptocurrencies of this form built from the ground up with NIPoPoWs in mind. Most notably, Ergo [6], Nimiq [29] and WebDollar [30].

1.3 Structure

In Chapter 2 we will outline some cryptographic primitives which are used by Bitcoin, NIPoPoWs or our implementations. We will explain how Bitcoin works and introduce our model for proofs. We will then explore NIPoPoWs and finally Velvet Forks.

Then in Chapter 3 we will investigate what assumptions we have to make in order to implement NIPoPoWs on Bitcoin Cash. We will focus on what changes have to be made and analyze our interlinker implementations.

In Chapter 4 we will explore how to generate proofs from our newly velvet forked chain. We will also analyze our prover implementation.

Finally, in Chapter 5 we will look at potential applications where our proofs can be used.

Chapter 2

Background

2.1 Primitives

2.1.1 Cryptographic Hash Functions

A *hash function* is a function used to map data of arbitrary size to data of a fixed size. Formally, a hash function H is of the form $H : D \rightarrow [0, 2^{\kappa})$, where κ is a characteristic of H, specifically the number of bits of its output.

For a hash function to be a cryptographic hash function it has to satisfy the following properties [21]:

- **Pre-image resistance**: Given a hash of h it should be difficult to compute an input m such that $H(m) = h$.
- **Second pre-image resistance**: Given an input m_1 it should be difficult to compute an input m_2 such that $H(m_1) = H(m_2)$.
- **Collision resistance**: It should be difficult to compute two inputs m_1 and m_2 such that $H(m_1) = H(m_2)$.

Bitcoin utilizes two hash functions, SHA256 ($\kappa = 256$) and RIPEMD160 ($\kappa = 160$).

This work and the works it is based on [18] operate within the random oracle model, where a hash function is assumed to be a truly random function [21].

2.1.2 Public-key Signatures

Each user is assumed to have a *key pair* composed of a *public key* which can be shared freely and a *private key* which should be kept secret. A public-key cryptographic system should implement the following operations for signatures [21]:

- $Sig_{sk}(m)$
- $Ver_{pk}(m)$ where $\forall m : \forall sig : Ver_{pk}(sig) = True \rightarrow sig = Sig_{sk}(m)$ and it is difficult for someone without pk to create such a sig

Bitcoin utilizes *ECDSA* [28], which is based on elliptic curves, as its public-key cryptography implementation.

2.1.3 Merkle Trees

A Merkle tree [24] is a data structure which allows a party to commit to a set of items using only a single hash, and prove the inclusion of any item in the committed set by providing a logarithmic proof in terms of the cardinality of the set.

More specifically, the hashes of the items consist the leafs of the tree, and the last level. The internal levels are defined recursively as follows: To create level $k-1$ each pair of level k (A, B) is

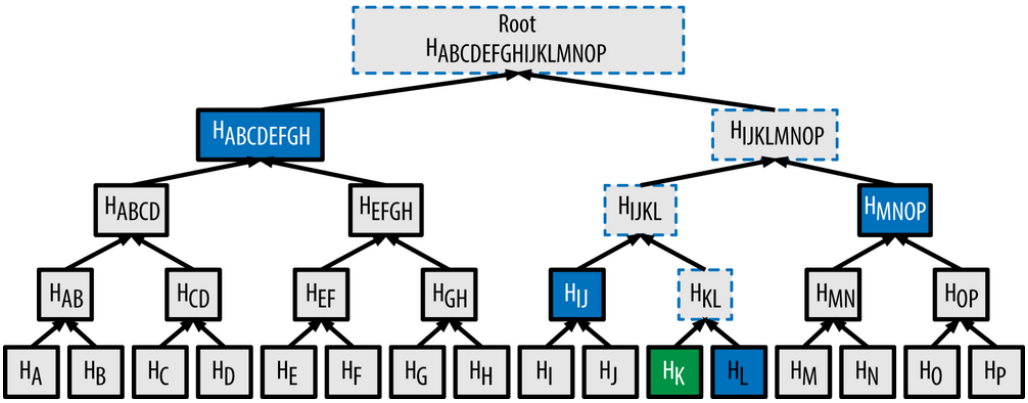

Figure 2.1: A Bitcoin Merkle tree. Source: [1]

transformed as a node of value $H(A||B)$ which points to both A and B. If the number of nodes at level k is odd, the last node at that level is paired with itself. [1]

Merkle trees are useful in Bitcoin in order to commit to a set of transactions to be included in a block while keeping the block header of a constant size as we will see shortly.

To provide proof of inclusion, all a prover has to do is provide a path of siblings up to the root siblings and a bit vector left indicating whether each sibling is on the left or the right. The verification process is shown in Algorithm 1.

Algorithm 1 The Verify algorithm for a Merkle proof

```
1:  function Verify_root(leaf, siblings, left)
2:      currentHash ← leaf
3:      while left ≠ [] do
4:          siblingIsLeft ← left.shift()
5:          if siblingIsLeft then
6:              currentHash ← H(siblings.shift()||currentHash)
7:          else
8:              currentHash ← H(currentHash||siblings.shift())
9:          end if
10:     end while
11:     return currentHash = root
12: end function
```

An example of a Bitcoin Merkle tree, along with a proof of inclusion for K can be seen on Figure 2.1.

2.1.4 Bloom Filters

A *bloom filter* is a probabilistic data structure allowing queries for set inclusion [4]. Specifically, a set of items is transformed to a bit vector size m. Starting from an empty bit vector of all zeroes, let us examine how we change the bit vector to indicate that an element is included in the set. Let us assume that we have k hash functions $h_1, \ldots, h_k$, where $\forall x \forall i \in [0, k] : h_i(x) \in [0, m-1]$.

The algorithm for item insertion seen in Algorithm 2 gets k values from all the hash functions on the new item, and then uses those values as indices. Every index indicates a bit that will be turned on in the new array. The new bloom filter is then returned.

[1] This specific construction is the one Bitcoin implements. There are various other constructions which are not inside the scope of this paper.

`1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`

Figure 2.2: A Bitcoin address

```
OP_HASH256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
OP_EQUAL
```

Figure 2.3: A Bitcoin script

Algorithm 2 The Insert algorithm for a Bloom Filter.

```
1: function INSERT(vec, item)
2:     for i ← 1 to k do
3:         vec[h_i(item)] ← 1
4:     end for
5:     return vec
6: end function
```

Testing a bloom filter for inclusion of an item is equally easy, as seen on Algorithm 3. The item again passes through the k hash function and the algorithm returns true if and only if all the indices pointed out by the k outputs are 1 bits on the array. It should be obvious however that there may be false positives, which is why it is necessary for an extra step in case one needs to be certain that the item is really included in the set.

Algorithm 3 The Query algorithm for a Bloom Filter. May return false positives.

```
1: function QUERY(vec, item)
2:     flag ← 1
3:     for i ← 1 to k do
4:         if vec[h_i(item)] ≠ 1 then
5:             return False
6:         end if
7:     end for
8:     return True
9: end function
```

2.2 Bitcoin

Bitcoin was invented in 2008 by Satoshi Nakamoto [26] as a peer-to-peer version of electronic cash, allowing online payments to be sent directly between parties without the need of a intermediary.

Bitcoin is pseudonymous: the identity of each user is only their *address*, which corresponds to an ECDSA public key. This address can be used to receive money from other users. Each user can spend money only if they have their corresponding private key. A set of ECDSA keypairs comprises a *wallet*. A user can have multiple addresses.

Transfer of value in Bitcoin happens with transactions. A transaction has inputs and outputs. An output is where the value creation happens for the receiver. An output can be later redeemed by using its designated receiver's private key and turned into an input to be used for another transaction.

2.2.1 Scripts

Bitcoin offers much more than just moving currency around. It allows us to actually move currency conditionally, where the condition can be expressed as a *Bitcoin script*. Bitcoin script is a stack-based language [28]. An example of a Bitcoin script can be seen on Figure 2.3.

This is an interesting script because it introduces two kinds of operations. First is commands prefixed with `OP_` (called *opcodes*): these perform operations on values (usually the top 1 or 2) on the stack and push the result to the stack. Specifically, `OP_HASH256` calculates the SHA256 hash

of the value on the top of the stack (and pushes the result on the stack). `OP_EQUALS` compares the top 2 values on the stack and pushes 1 or 0 if they are indeed equal or not accordingly. Values like `6fe...` in hex with no `OP_` prefix are simply pushed to the stack.

So what does this script do? It checks if the value on the stack is the preimage of the given hash. If we do push the correct preimage in the stack before running the script, the result is going to be 1, letting us know that the evaluation was successful.

Such a script express our predicate, and is called a pubKeyScript. However the predicate must run on something. In our example above we assumed the preimage was on the stack before the stack ran, and this is how we parameterized our predicate. The way this is done in Bitcoin is running another script before the predicate called scriptSig, to essentially pass the parameters to our predicate. We'll see shortly how these can be leveraged to actually transfer funds.

It's important to note that there's a third type of opcodes which can fail the script preemptively. These opcodes are usually in the form of `OP_...VERIFY`. An example is `OP_EQUALVERIFY` which fails the script if the top 2 values of the stack are not equal, and is otherwise a no-op.

Next we'll look at a couple of interesting classes of scripts.

Anyone-can-spend

It's easy to make the predicate true for everyone. We can make pubKeyScript be `OP_TRUE` (which adds the true value to the stack) and scriptSig be empty.

P2PKH

This is the standard script for conventional fund transfer in Bitcoin. Let's say we want to make sure only Bob can satisfy this script. The pubKeyScript is the following: `OP_DUP OP_HASH160 <Bob's address> OP_EQUALVERIFY OP_CHECKSIG`.

The scriptSig is then typically `<Bob's signature> <Bob's public key>`. Values in `<>` are placeholders for actual values, they're not valid script commands. These have to be replaced with concrete values before the above scripts can be valid.

Bob's signature will be on the hash of the transaction (which we'll explore shortly) containing the output. The script will then duplicate his public key, check that it matches the one on the pubKeyScript, and if it does check that he has provided a valid signature with that public key. If all these checks pass the stack will end up with 1 on top and the execution will be valid, thus the predicate will be satisfied.

OP_RETURNs

It frequently is desired to add arbitrary data to the blockchain (e.g. for timestamping). To do this, one can make the arbitrary data part of the pubKeyScript. There's a special opcode for such cases called `OP_RETURN` which can be followed by a series of arbitrary data (in hexadecimal). `OP_RETURN` fails the script preemptively so no `scriptSig` can satisfy it. We'll see how based on `OP_RETURN`s we can greatly augment the functionality of Bitcoin later on.

2.2.2 Outputs

An *output* is a tuple (value, pubKeyScript). The value refers to an amount of Bitcoin in Satoshi (where 10^8 Satoshi = 1 Bitcoin) and pubKeyScript is a script which needs to be run against some stack and return 1 in order for value to be transferable.

2.2.3 Inputs

An input is the way an output is redeemed. Specifically, it contains 3 things:

- The hash of the transaction where the output of interest is contained.
- An index clarifying which output in the transaction this input is referring to.
- A signature (called scriptSig) used for the validation of the output script.

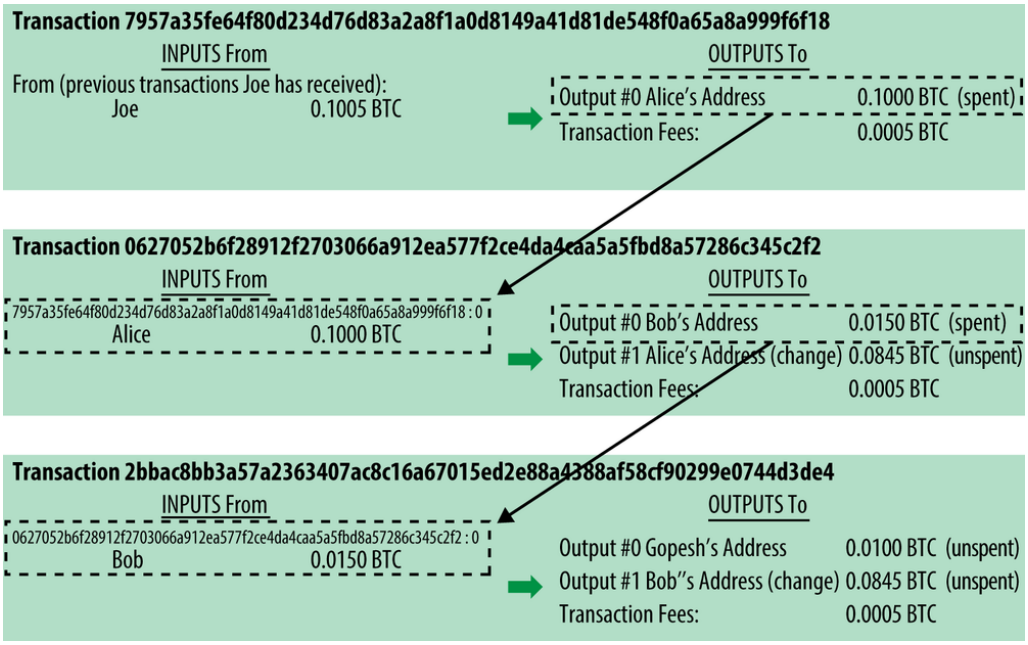

Figure 2.4: Transactions with their inputs and outputs. Source: [1]

For the transaction to be valid, the script inside the specified output when run on a stack with the contents of scriptSig should return 1.

As a convention, when we talk about the value of an input we mean the value of the output it redeems.

2.2.4 Transactions

A *transaction* is a collection of inputs and outputs. It uses the sum of the inputs values' as credit to debit each output accordingly. As it makes sense, a transaction is only valid as long as all its outputs and inputs are valid. It should also be clear that the value of the outputs should not exceed the value of the inputs, otherwise we would be creating value out of thin air with new transactions. Specifically this is expressed as $\sum_{\mathsf{i}\in\mathsf{inputs}} \mathsf{i.value} \geq \sum_{\mathsf{o}\in\mathsf{outputs}} \mathsf{o.value}$. This is sometimes called the *Law of Conservation*.

In cases where $\sum_{\mathsf{i}\in\mathsf{inputs}} \mathsf{i.value} > \sum_{\mathsf{o}\in\mathsf{outputs}} \mathsf{o.value}$ we call

$$\sum_{\mathsf{i}\in\mathsf{inputs}} \mathsf{i.value} - \sum_{\mathsf{o}\in\mathsf{outputs}} \mathsf{o.value}$$

the *transaction fee*. This is paid to the miner who successfully mines a block containing the transaction. This is one of the two ways Bitcoin uses to incentivize miners.

2.2.5 Blocks

A block contains a list of transactions, the first of which is called the *coinbase transaction* which is where value creation happens in Bitcoin. The miner crafts this transaction granting them some amount of Bitcoins and this transaction is going to be valid only if the block turns out valid. This doesn't mean that anyone can generate Bitcoin out of thin air: we'll see shortly how it actually comes at a cost with Proof-of-Work.

A block header contains mainly the hash of the previous block, a Merkle root hash to commit to a set of transactions, and a nonce. Blocks are always referenced by the hash of their block header.

Once a transaction has been included in a valid block it's called *confirmed*.

It's possible that there are contending chains of blocks. We then say there is a *fork* on the chain. On Figure 2.6, the chain has forked on blocks 3 and 6.

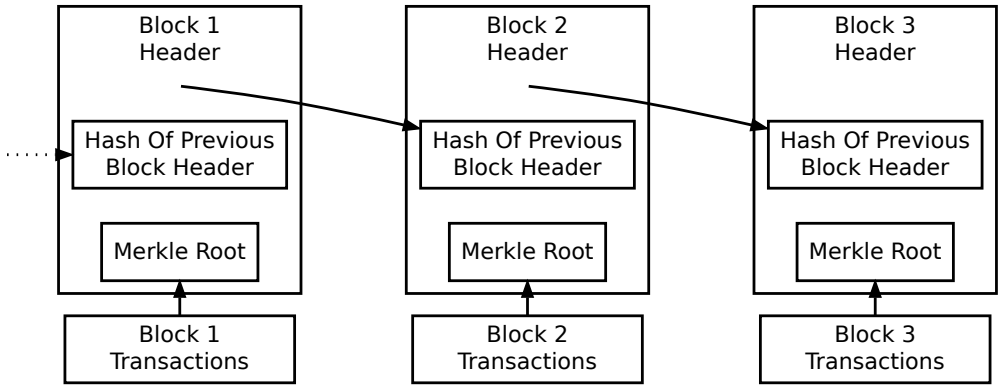

Figure 2.5: The block structure. Source: [26]

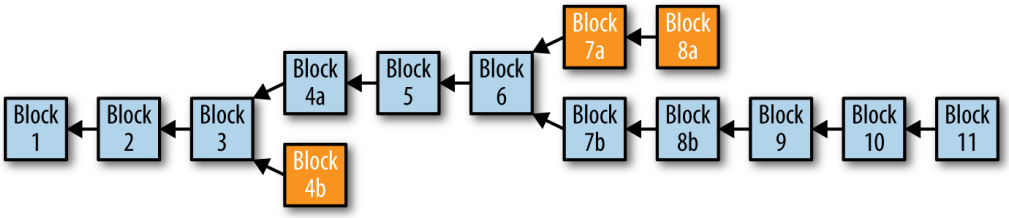

Figure 2.6: A block chain. The orange blocks are orphans. Source: [1]

We call any valid blocks which are not part of our active chain *orphans*. For example, on Figure 2.6, blocks 4b, 7a and 8a are orphans.

2.2.6 Proof-of-Work

The key to making Bitcoin decentralized is a technique called Proof-of-Work. Proof-of-Work was first invented in 1992 by Dwork et al. [9] as a measure of limiting email spam and denial-of-service attacks and later explored by Back [3] as Hashcash.

We'll examine a simplified model of Hashcash in order to explore the idea. Suppose we want to send an email to someone. In order to prove we've done work, we include a header (like `X-Hashcash`), which (very simplistically) includes the receiver's email address, and a nonce. [2] The nonce is picked so that the hash of the header $H(email||nonce)$ has its 20 most significant bits be all 0. The only feasible way to find this is by brute-forcing the nonce. Once the sender has found the nonce it's included in the header and sent.

The receiver can then very easily check whether the header hashes to a valid value. If that's so, the email it contains belongs to the receiver and the header is not being reused, then the email can be considered not spam.

To reiterate, the idea is having a series of data to commit to and a hole for the nonce, which is brute-forced to satisfy a necessary predicate on the hash, specifically that its n most significant bits are all zeroes. This is exactly how Bitcoin implements Proof-of-Work. Instead of the hole being on an email header the hole is on the block header. For a block to be valid, its header has to satisfy a predicate like the above.

Bitcoin introduces a couple of differences. n varies according to the block generation rate. Specifically, to translate the previous predicate to Bitcoin terminology, the hash of each block header has to satisfy $H(\text{blockHeader}) \leq T$ where T is called the *target*. As the target goes up, the probability of being below it goes up and generating a valid block is easier. Conversely, if the

[2] Hashcash headers actually contain 7 different fields which have been omitted here for simplicity. The simplified version explained here is not making the same security guarantees as Hashcash.

target goes down it's harder to generate a valid block. To express this, in Bitcoin $\frac{1}{T}$ is called the *difficulty*.

To account for the block generation rate, which Bitcoin tries to keep to 1 block per 10 minutes, every 2016 blocks the target (and subsequently the difficulty) is adjusted accordingly. The target is calculated inside the Bitcoin software and is only as a function of the blocks previously seen (frequently called their *view*), so as long as the Bitcoin nodes agree on the view they'll agree on the target and all consider the same set of incoming blocks as valid.

2.2.7 Simplified Payment Verification

The size of the blockchain has reached 185GB by September 2018, which makes it a very time consuming or even infeasible process to synchronise a full node. Fortunately, a solution was proposed in the original whitepaper [26], which allows the creation of so-called *lite nodes*. Lite nodes only know the headers of the entire blockchain, which are constant-size for each block (80 bytes). At the time of writing of this paper, the size of all block headers was ∼42MB. The lite node then asks the network for transactions concerning it (e.g. transactions concerning a specific public key). Full nodes of the network find such transactions and return them to the requester. For each transaction, the block header of the block it's included in is returned, along with a Merkle tree proof of inclusion which the lite node can then verify. This protocol is reliable as long as an adversary does not control the network of a lite node.

2.2.8 Bitcoin Cash

In 2017 Bitcoin faced severe scalability issues [8]. Its limited 1MB block size meant that it could only support a maximum of 7 transactions per second. As Bitcoin's popularity had exploded at the time, the problem was hugely exacerbated. The most prominently proposed solution for this was a block size increase, however no consensus was reached. The discussions ended with a fork of the main Bitcoin chain which allowed for 8MB blocks, called Bitcoin Cash.

2.3 The Prover-Verifier Model

Before we talk about the specifics of proofs we first have to define our setting. In our setting we have two kinds of actors, *provers* and *verifiers*.

A verifier is a party who wishes to know something about our blockchain. It's assumed it doesn't have network access (i.e. it can't be a full node). A verifier can be thought of as a Turing Machine which takes one or more proofs as input, and then determines whether a predicate is true or not. The only information the verifier knows is the genesis id of the chain it wants to decide predicates on.

Provers are parties with access to the Bitcoin network, the longest chain and all the transactions. They wish to communicate to the verifier and convince them about a predicate.

We distinguish provers as honest and malicious. An honest prover will generate proofs which are true according to the state of the chain he is aware of. A malicious prover may try to submit false proofs in order to gain some advantage. For example, a verifier may be a merchant, who, upon confirming that his client, Bob, has paid for a product will ship it to them. In this case it is in the best interest of Bob to submit a malicious proof which makes it appear like he has paid for the product when in reality he has not.

2.3.1 Single-Prover Proofs

There are many interesting statements a verifier can be certain about by utilizing only a single prover. We say that a chain is *valid* iff it is structurally sound, which we also call *traversable*.

Chain Validity Proof

Suppose we have a chain $\mathcal{C}$ and we wish to prove to the verifier that it is a valid chain. It is easy to do this by supplying the whole chain to the verifier, who can sequentially, starting from the newest block, verify that the previous block hashes to the hash found on the newest block's previd. The

process can be repeated until the genesis block is reached where the verifier can accept the chain as valid.

Transaction Inclusion In Selected Block Proof

If we have a transaction tx which we wish to prove exists in a block B, then we can just generate a Merkle proof for the inclusion of the transaction's id. Given B, tx and the Merkle proof, the verifier can then verify the Merkle proof against the Merkle tree root inside the header of B to confirm that $tx \in B$.

Transaction Inclusion In A Valid Chain Proof

In order to prove that a transaction tx is included in some valid chain we can very easily combine the above proofs. Suppose that $tx \in B$. Then by providing a chain validity proof for the chain that contains B and a transaction inclusion proof for B, the verifier can accept if and only if both proofs are valid and the chain provided includes B.

It is important to note that our definition of validity is less strict than a full node's definition of validity. As a full node, it is possible to check that every transaction in a block is valid, and if and only if this is true, and the block is well-formed and valid under Proof-of-Work consider a block valid.

2.3.2 Multi-Prover Proofs

However, while a single prover is enough for some types of proofs, some are impossible to achieve without multiple provers. For example, convincing a verifier that we hold the longest Bitcoin chain would be impossible to do, because a prover could very easily be malicious and present a shorter chain. Without taking extra input from other provers and assuming that there exists at least one honest prover, it is impossible for a verifier to be certain. Those are two assumptions we will make for the following proofs.

Longest Chain Proof

Suppose we have a chain $\mathcal{C}$ and we wish to prove to the verifier that it is the longest chain. We submit the whole chain to the prover, as do the rest of the provers. The verifier then verifies each given chain for validity and discards any invalid chains. For the remaining chains it compares them and keeps the longest one.

Transaction Inclusion In The Longest Chain

As previously, suppose $tx \in B$. We provide the whole chain, along with the Merkle proof for $tx \in B$. The verifier collects proofs of this kind and selects the longest valid chain. Then for all transaction inclusion proofs taken it checks whether the claimed block is included in the longest chain and the transaction inclusion proof is valid. If there is at least one transaction inclusion proof satisfying this requirement then the transaction is accepted, otherwise it is rejected.

Verifying this kind of proof is very similar to the operations a lite node does to verify transactions claimed to be on the longest chain that we viewed in Section 2.2.7, which is why these proofs are called SPV proofs. These proofs are all linear in the size of the chain.

2.4 Non-Interactive Proofs of Proofs of Work

Seeing that proofs are linear in the chain length, it is natural to consider if we can do better. NIPoPoW [18] provides the first family of succinct proofs which are logarithmic in the size of the chain and proven secure in the Backbone [10] protocol.

There have been previous attempts to create proofs smaller in size than SPV proofs [17], where a scheme for logarithmic proofs was proposed. This scheme was later proven insecure [18].

T	1110000
id(B)	0001000

Table 2.1: Calculating the level of a block by counting the leading zeros (3 in this case).

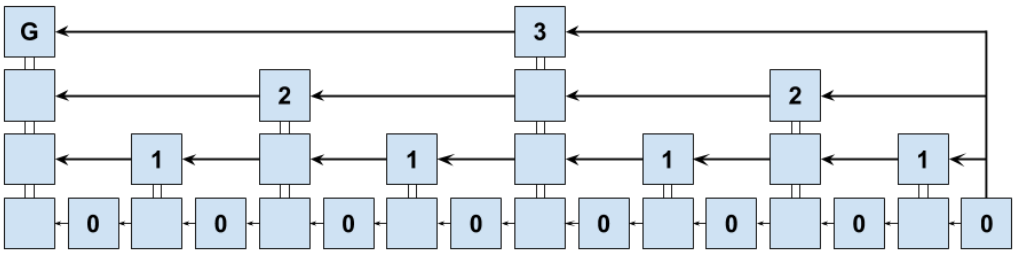

Figure 2.7: The hierarchical blockchain. Higher levels have achieved a lower target (higher difficulty) during mining. All blocks are connected to the genesis block G. Source: [18]

2.4.1 Terminology

NIPoPoWs are categorized in two kinds of proofs:

- We have a valid chain where the last k blocks (also called the *unstable* part of the chain) are the ones we're claiming. This is called a **suffix proof**.
- We have a valid chain where a specific given block is included in its *stable* part (excluding the last k blocks). This is called an **infix proof**.

2.4.2 Assumptions

An assumption NIPoPoWs make is that the difficulty is constant. This is not true for Bitcoin or Bitcoin Cash.

NIPoPoWs also assume each block contains an interlink data structure, which we'll study shortly. Interlinks too don't exist in Bitcoin or Bitcoin Cash.

In the next section we'll look at how we sidestep all those issues.

2.4.3 Levels

At the heart of the primitive lies the separation of blocks into levels. The level of a block is defined as $level(B) = \lfloor \log(T) - \log(\mathsf{id(B)}) \rfloor$, where T is the constant difficulty of the blockchain. The genesis block is an exception to this rule as $level(Gen) = \infty$. We call a block of level μ a μ-superblock.

Intuitively, the level of a block is the number of leading zeros of the binary representation of the block id when left padded to the length of T. An example of this can be seen on Table 2.1.

Figure 2.7 shows an example the implied blockchain created from the superblocks.

2.4.4 Notation

The NIPoPoWs paper introduces some notation for talking about blockchains with levels which we'll be using extensively. The notation is widely influenced by Python. Specifically:

- $\mathcal{C}$ denotes a blockchain, with $\mathcal{C}[0]$ being the genesis block, $\mathcal{C}[k]$ being the k-th first block and $\mathcal{C}[-k]$ being the k-th last block.
- $\mathcal{C}[k:]$ denotes the sub-blockchain starting from the k-th block, $\mathcal{C}[-k:]$ denotes the sub-blockchain starting from the k-th last block.
- $\mathcal{C}[:k]$ denotes the sub-blockchain ending before the k-th block, $\mathcal{C}[:-k]$ denotes the sub-blockchain ending before the k-th last block.

Level	Block
0	$C[-2]$
1	$C[-2]$
2	$C[-4]$
3	$C[-8]$
∞	$C[0]$

Table 2.2: Interlink of $C[-1]$ from Figure 2.7

- $\mathcal{C}[i:j]$ denotes the sub-blockchain starting from the i-th block and ending at the j-th block. i and j can also be negative numbers similar to above.
- $\mathcal{C}\{B:\}$ denotes the sub-blockchain starting from the block with block id B.
- $\mathcal{C}{\uparrow}^{\mu}$ denotes the sub-blockchain of $\mathcal{C}$ where all blocks are of level μ or higher.

2.4.5 Interlink

Instead of keeping only the hash of the previous block inside the block header, for every superblock level we keep a pointer to the most recent superblock of that level. The structure containing these pointers is called the interlink. Bitcoin does not support such a structure in the block header but we will study how to sidestep this issue by velvet forking in a few sections.

It's important to note that the interlink can be encoded as a series of block ids, starting from 0 up to ∞. It can also be compressed by using this series as the leafs of a Merkle tree and taking the Merkle tree root.

Suppose we have a block B' with an interlink stored as $B'.\mathsf{interlink}$. In order to produce the interlink for a block after B' we make sure to change all pointers from level 0 up to $level(B')$ to point to B', as B' will be the most recent block at these levels (remember that a block of level μ is also of level $\mu - 1$). We call this procedure updateInterlink, which can be seen in detail on Algorithm 4.

Algorithm 4 updateInterlink

```
1: function updateInterlink(B')
2:     interlink ← B'.interlink
3:     for μ = 0 to level(B') do
4:         interlink[μ] ← id(B')
5:     end for
6:     return interlink
7: end function
```

2.4.6 Suffix Proofs

Suffix proofs are parameterized by k and m. k refers to the number of blocks that need to bury a block for it to be considered stable.

A suffix proof of a chain $\mathcal{C}$ is constituted of two chains, π and χ. The final proof is the concatenation of those two chains $\pi\chi$. χ always refers to the chain of unstable blocks and is evaluated as $\chi = \mathcal{C}[-k:]$.

The process for constructing π is a little more convoluted. First we have to find the first level μ where $|\mathcal{C}{\uparrow}^{\mu}| \geq m$. We call this level $max\mu$. For this level we take all its blocks except for the last m: $\pi_{max\mu} = \mathcal{C}{\uparrow}^{\mu}\ [:-m]$. Then for every level μ from $max\mu - 1$ to 0, we take blocks $\pi_{\mu} = \mathcal{C}{\uparrow}^{\mu}\ [:-m]\{\mathcal{C}{\uparrow}^{\mu+1}\ [-m]:\}$.

π is then evaluated as the concatenation of all those chains starting from the oldest block:

$$\pi = \pi_{max\mu}||\pi_{max\mu-1}||\dots||\pi_0$$

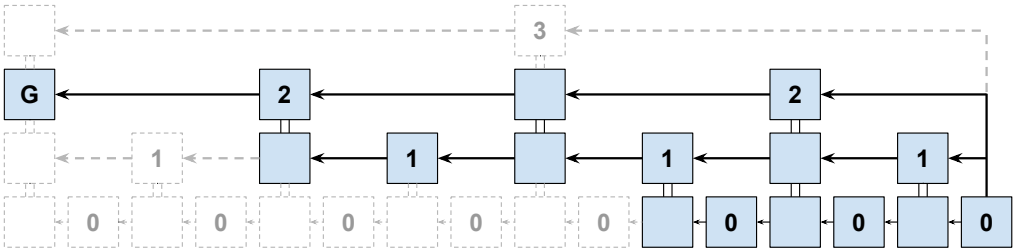

Figure 2.8: Construction of π of a suffix proof. $m = 3$ Source: [18]

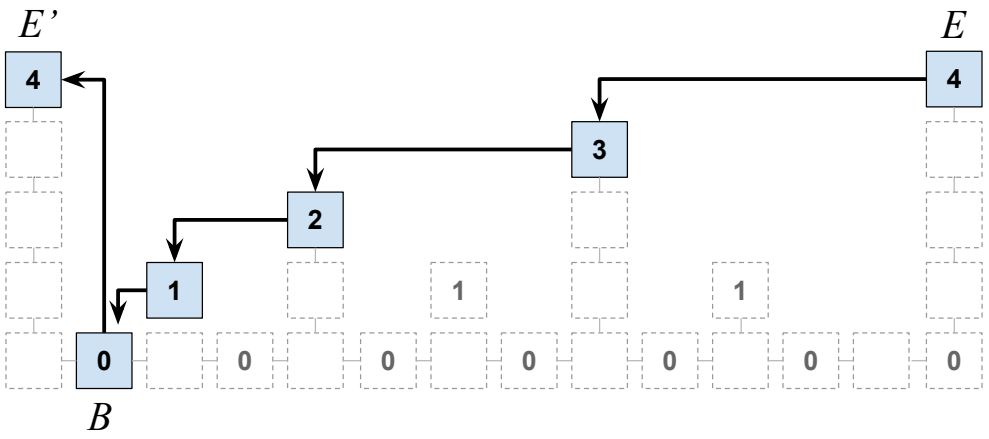

Figure 2.9: Construction of an infix proof. Source: [18]

It's important to notice that the chain provided to the verifier is an actual chain: one can start at the end and traverse it until the genesis block by utilizing the interlink of each block, similar to how they would do that on a conventional blockchain by using each block's prevId.

2.4.7 Infix Proofs

For the verifier to be able to determine a predicate on one or more blocks ($\mathcal{C}' \subseteq \mathcal{C}$) of our chain, we have to make sure we include them in a proof. A suffix proof is not guaranteed to include all blocks of interest in $\mathcal{C}'$. In order to include these we have to make sure they are linked to the proof, e.g. that the proof chain is a traversable. To this end, let's assume some arbitrary block $B \in \mathcal{C}'$. Let's also assume an existing suffix proof $\pi\chi$. We find blocks E' and E on the suffix proof, such that:

- E is the next block after E' on the proof
- B comes before E on $\mathcal{C}$
- B comes after E' on $\mathcal{C}$

An example of such a triplet of blocks satisfying those conditions can be seen on Figure 2.9.

We then perform a procedure called followDown in order to figure out which blocks need to be added to the proof in order to link E to B. followDown includes blocks on intermediate levels until B is reached. The full algorithm can be seen on Algorithm 5.

Algorithm 5 The followDown function which produces the necessary blocks to connect a superblock E to a preceeding regular block B. Source: [18]

```
1: function followDown(E, B, height)
2:     aux ← ∅; μ ← level(E)
3:     while E ≠ B do
4:         B' ← blockById[E.interlink[μ]]
5:         if height[B'] < height[B] then
6:             μ ← μ − 1
7:         else
8:             aux ← aux ∪ {E}
9:             E ← B'
10:        end if
11:    end while
12:    return aux
13: end function
```

Augmenting the original suffix proof with the new blocks provided by followDown on all our blocks of interest $B \in \mathcal{C}'$ gives us our final infix proof. Note that as was the case with our suffix proofs, the infix proof is traversable.

2.4.8 Proof Validation

It is very easy to validate a proof, similarly to how we validated SPV proofs in the previous chapter: one can just check that the proof is traversable up to a known genesis, and $|\chi| = k$. Validation in the context of NIPoPoWs refers to a single-prover environment.

2.4.9 Suffix Proof Verification

In a multi-prover environment, we noted the need for a way to compare proofs. For SPV proofs this was easy, the length of each chain could be compared (as long as both chains were valid). For NIPoPoWs however the process of comparing is not as straightforward, if we consider that the number of blocks is not directly connected to chain length.

In Algorithm 6 we show how two proofs can be compared. The algorithm finds the Lowest Common Ancestor b from the stable part of both proofs, and then calculates best-arg on each proof from b onwards. The proof with the largest best-arg is decided to be the better proof.

Algorithm 6 The algorithm implementation for the $\geq_m$ operator to compare two proofs in the NIPoPoW protocol parameterized with security parameter m. Returns *true* if the underlying chain of player A is deemed longer than the underlying chain of player B. Source: NIPoPoWs paper [18]

```
1: function best-arg_m(π, b)
2:     M ← {μ : |π↑^μ {b :}| ≥ m} ∪ {0}                ▷ Valid levels
3:     return max_{μ∈M}{2^μ · |π↑^μ {b :}|}            ▷ Score for level
4: end function
5: operator π_A ≥_m π_B
6:     b ← (π_A ∩ π_B)[−1]                             ▷ LCA
7:     return best-arg_m(π_A, b) ≥ best-arg_m(π_B, b)
8: end operator
```

Knowing how to compare two proofs, the only thing a verifier has to do is actually compare all the proofs it receives (which are collected in the set $\mathcal{P}$) and only keep the best one. This is shown in Algorithm 7. As an extra step, after picking the best proof $(\tilde{\pi}, \tilde{\chi})$, this verifier for suffix proofs evaluates a predicate $\tilde{Q}$ on the suffix, or unstable part of the best proof it received.

Algorithm 7 The Verify algorithm for the NIPoPoW protocol. Source: NIPoPoWs paper [18]

function $\mathsf{Verify}^{Q}_{m,k}(\mathcal{P})$
  $\tilde{\pi} \leftarrow (\text{Gen})$ ▷ Trivial anchored blockchain
  for $(\pi, \chi) \in \mathcal{P}$ **do** ▷ Examine each proof (π, χ) in $\mathcal{P}$
    if $\mathsf{validChain}(\pi\chi) \wedge |\chi| = k \wedge \pi \geq_m \tilde{\pi}$ **then**
      $\tilde{\pi} \leftarrow \pi$
      $\tilde{\chi} \leftarrow \chi$ ▷ Update current best
    end if
  end for
  return $\tilde{Q}(\tilde{\chi})$
end function

2.4.10 Velvet Forks

Velvet forks [18,33] describe a formalization of adding arbitrary data inside blocks in order to allow potential applications without sacrificing the backwards compatibility of the blockchain.

Miners who are willing to contribute to the fork can add data of interest in the form of coinbase transaction data.

Backwards compatibility is achieved by not changing the consensus rules, meaning that set of acceptable blocks does not change. So any block that was acceptable remains acceptable even if it does not contain any data concerning the fork, or if it contains invalid data.

2.4.11 User-Activated Velvet Forks

In case miners aren't interested in including such data, users can also create such a fork by making a kind of transaction called velvet transaction. In a velvet transaction a user includes any data of interest in unspendable transaction outputs (like OP_RETURN).

For the consumer of such data, the only difference is that they have to look inside the whole block to find it, not only inside the coinbase data.

Such forks come at the cost of making such transactions, because the user who makes the fork needs to pay transaction fees every time they wish to add data to the blockchain.

2.4.12 Velvet NIPoPoWs

Since anyone can post such transactions on the blockchain, we have to make sure that the commitment is actually true before we can use it. In order to do that we maintain our own version of the interlink for each block which we know is correct called realLink. Then for every block, we compare its commitments (there may be many) with the Merkle Tree root of our realLink. If there is a valid commitment we say that the block has a valid interlink. We store the full interlink as realLink[id(block)].

We already know how it's essential that our proof forms a blockchain that can be traversed from start to end. In order to make our proof traversable, whenever we include a block we have to make sure it connects validly to the previous one either by (a) using the regular previd inside the block or (b) using a valid interlink. If we use the previd to link back to the previous block then all the information someone needs to verify the traversability is already there and we don't need to add anything extra. In the case we use the interlink however, we need to provide the Merkle Tree proofs for:

- The transaction containing the valid interlink commitment.
- The interlink level we use for the connection.

If our chain is not traversable the proof is automatically invalid.
We'll now look at the concrete implementation of the prover.

2.4.13 Invalid Interlink Handling

The original NIPoPoWs paper [18] gives us some insight into how to handle blocks with invalid interlinks. Let's look at where we need to make changes starting with suffix proofs.

For suffix proofs we need a way to obtain the upchain of a chain, denoted as $\mathcal{C}\uparrow^{\mu}$. We will now define a procedure to programatically obtain the upchain of a chain called $\mathsf{find}\mathcal{C}\uparrow^{\mu}$.

followUp

followUp takes a block b and a level μ as parameters. Starting from b it traverses the chain until it reaches another block of level μ called B. In doing so, it is only allowed to use valid pointers. It will only follow a pointer from a block's interlink at level μ if there is a valid interlink in that block. Otherwise, the only option is to follow the previous block pointer (previd). Once B is reached, it is returned alongside the blocks that were traversed as a blockset called aux.

Algorithm 8 followUp produces the blocks to connect two superblocks in velvet forks.

```
1:  function followUp(B, μ, realLink, blockById)
2:      aux ← {B}
3:      while B ≠ Gen do
4:          if B.interlink[μ] = realLink[id(B)][μ] then
5:              id ← B.interlink[μ]
6:          else                                          ▷ Invalid interlink
7:              id ← B.previd
8:          end if
9:          B ← blockById[id]
10:         aux ← aux ∪ {B}
11:         if level(B) = μ then
12:             return B, aux
13:         end if
14:     end while
15:     return B, aux
16: end function
```

find$\mathcal{C}\uparrow^{\mu}$

Now that we have a way to go back on a level, we can utilize it to construct the entire traversable level μ up to block b, starting from the end of the chain $\mathcal{C}[-1]$: this is what $\mathsf{find}\mathcal{C}\uparrow^{\mu}(b)$ accomplishes. It works by repeatedly calling followUp on the oldest block it has and including the result in its final chain.

Algorithm 9 Supplying the necessary data to calculate a connected $\mathcal{C}\uparrow^{\mu}$ during a velvet fork.

```
1:  function find C↑^μ(b, realLink, blockById)
2:      B ← C[−1]
3:      aux ← {B}
4:      π ← [ ]
5:      if level(B) ≥ μ then
6:          π ← πB
7:      end if
8:      while B ≠ b do
9:          (B, aux') ← followUp(B, μ, realLink, blockById)
10:         aux ← aux ∪ aux'
11:         π ← πB
12:     end while
13:     return π, aux
14: end function
```

Velvet Infix Proofs

The infix prover also changes to account for invalid or nonexistent interlinks. On followDown we have to make sure that any interlinks we use are valid. Thus, for any block with an invalid interlink we are forced to use its previd instead to find its previous block, in hopes that the previous block will contain a usable interlink.

Chapter 3

Velvet Fork Implementation

Now that we have seen how NIPoPoWs work and what options we have in our disposal to deploy them to existing blockchains, we will investigate how we implemented NIPoPoWs on Bitcoin Cash.

Since Bitcoin Cash blocks don't contain the interlink we have to utilize a velvet fork. We could choose a regular velvet fork. A regular velvet fork would require for a miner minority to exist and run code which would add the interlink inside the coinbase transaction. Finding and contacting miners would require a lot of non-technical work and persuasion. It would also mean that probably only a minority of the blocks would be validly interlinked. Seeing that persuading miners would require a lot of work and would not bring an ideal outcome we decided to implement a User-Activated Velvet Fork instead.

To this end, we need to make sure that a transaction is included in every single block containing its implied interlink. We do this by implementing a service which for every new block, calculates the expected interlink of the upcoming block and sends a transaction including this interlink in hopes that it will be included in the upcoming block. If this is indeed achieved then that block will indeed contain its valid interlink. We henceforth call the service which does this an *interlinker*. An example of blocks and their corresponding interlinks can be seen on Figure 3.1.

3.1 Picking a Velvet Genesis

Naturally one would expect the interlink to start from the real blockchain genesis as it would enable proofs for any already existing block of that blockchain. However, for older blocks there can be no improvement. There is no other option than providing the full chain as a proof since no older blocks contain any interlinks. Thus in order to avoid accounting in our interlink for blocks in the past that can only be connected to the real genesis by supplying the full chain we choose a new genesis called the *velvet genesis*. For our purposes in Bitcoin Cash testnet we chose the block with height 1257603 as our velvet genesis.

3.2 Faking Constant Difficulty

An issue with NIPoPoWs we mentioned early on, which makes them incompatible with Bitcoin is that a constant difficulty setting is assumed. Bitcoin and Bitcoin Cash, like most well-known cryptocurrencies have variable difficulty. Although research for NIPoPoWs in the variable difficulty setting is currently underway, no clear solution exists yet. For our purposes we treat the variable difficulty blockchain as if it was a constant difficulty chain with $T_{const} = \mathsf{powLimit}$, where $\mathsf{powLimit}$ is the maximum target that can be achieved in the variable difficulty chain. Assuming such a target means that all the valid blocks under variable difficulty are also valid under the fake constant difficulty, since $\forall B : id(B) \leq T \leq \mathsf{powLimit} = T_{const}$. [1]

It is trivial to determine $\mathsf{powLimit}$ for any cryptocurrency, as it is hardcoded in the source code of any full node. Here is an example from the Litecoin source code.

[1] This is the most reasonable construction according to the NIPoPoWs paper authors. We do not claim or prove that this construction is secure.

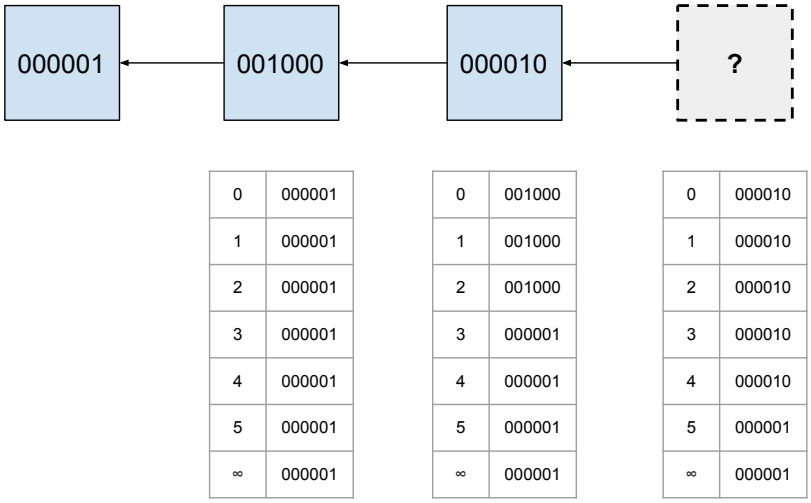

Figure 3.1: Example of blocks with their corresponding interlinks for $T = 100000$. Note that for the upcoming block marked with **?** we can produce its interlink without knowing its hash.

```
class CMainParams : public CChainParams {
public:
    CMainParams() {
        strNetworkID = "main";
        // ...
        consensus.powLimit =
             uint256S("00000fffffffffffffffffffffffffffffffffffffffffffffffffffffffffff");
        // ...
     }
}
```

3.3 Interlink encoding

We now have to consider our options for representing the interlink. Bitcoin Cash uses SHA-256 for the block hashes, meaning that each block id consists of 32 bytes. A naive encoding would be one where each 32-byte block hash from level 0 up to ∞ is concatenated, with the ∞ pointer always pointing to the genesis block.. For 35 levels, which according to Figure 3.2 is a reasonable figure, this encoding would take approximately 1.12KB. Considering that `OP_RETURN` scripts are limited in size to 223 bytes, this would only allow us to include up to 6 pointers at best.

Putting this limitation aside, the fee of the transaction is proportional to the transaction size, and since we're going to be sending a transaction for every block (which is mined approximately every 10 minutes), it is important that the fee is minimized.

Thus in order to save space, we only include a commitment to the interlink in our transactions. Specifically, we take the Merkle Tree root of the Merkle Tree with leafs the block hashes starting from level 0 up to ∞. This way, our interlink encoding is constant size and we can easily provide compact proofs for any of the levels.

3.4 Discoverability

We've talked about how just including the interlink somewhere on a block is what really matters but it is crucial that we make this information easy to discover. We achieve this in two ways. First,

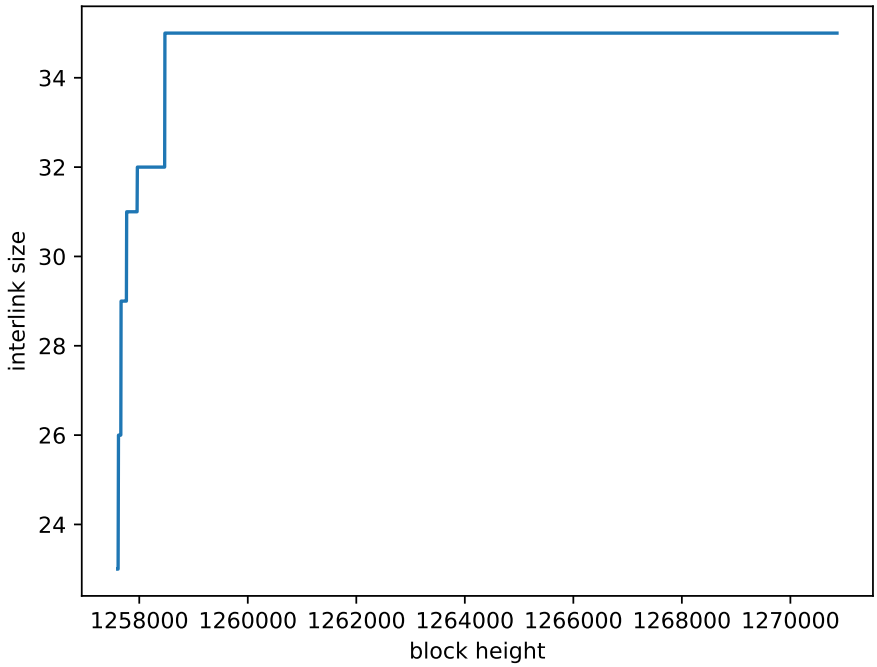

Figure 3.2: Number of levels assuming our selected velvet genesis.

we include the interlink commitment inside a special `OP_RETURN` output. Such outputs are already being used for storing arbitrary data in blocks [22] therefore we adopt this method for storing our interlinks. Second, we aim to make this interlink discoverable for lite nodes, so we don't require our users to download a whole block in order to look into it. We achieve this by utilizing a method called *SPV tagged outputs* [13].

SPV tagged outputs are outputs that are tagged so that they can be discovered by SPV nodes who add the "tag" to the filter. Specifically, we form outputs of the form `OP_RETURN baba deadbeef`, where `baba` is our tag and `deadbeef` is our payload (in our case, the interlink commitment). A bloom filter for `baba` will then match this output and subsequently the transaction that contains it and this is how our specialized SPV nodes can discover our outputs. The full nodes forwarding the velvet transactions to the SPV nodes don't have any knowledge of what a velvet fork even is, let alone that they are forwarding its transactions.

The tag we use for our transactions is `696e7465726c696e6b`, which is the ASCII encoding of `interlink`. An example of such a real-world velvet transaction created by our deployed interlinker on the Bitcoin Cash testnet can be seen on Figure 3.3.

3.5 Fault Tolerance

It is important to note that the interlinker works on a best-effort basis. Due to the nature of Velvet Forks though, no failure is fatal. The types of failures are as follows:

- **Crash failure**: The interlinker crashes or halts.
- **Omission failure**: The interlinker fails to push a transaction upon seeing a new block.
- **Timing failure**: The interlinker pushes a transaction which fails to be included in the upcoming block.
- **Response failure**: The interlinker pushes a transaction with an invalid interlink.

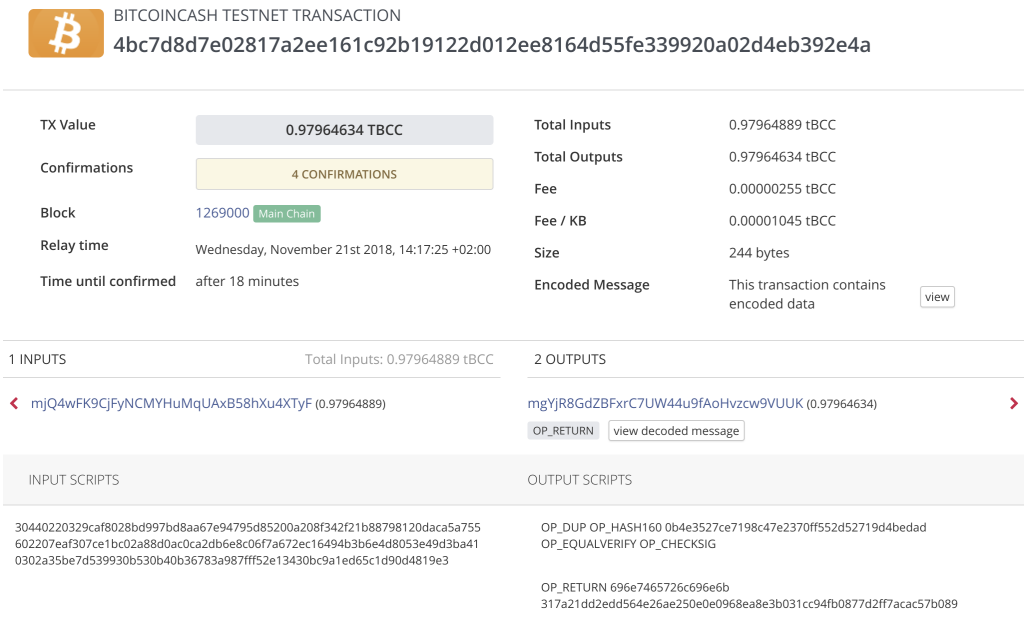

Figure 3.3: An actual velvet fork transaction.

Cost per	Formula	Current estimation
Day	$\mathsf{txFee} * 10 * 24$	0.13€
Month	$\mathsf{perDay} * 30$	3.9€
Year	$\mathsf{perMonth} * 12$	46.8€

Table 3.1: Cost analysis of operating an interlinker on the Bitcoin Cash mainnet.

- **Arbitrary failure**: The interlinker pushes a transaction before a new block is seen or pushes a duplicate for a specific upcoming block twice.

We will study how these failures can be mitigated in the next section.

3.6 Viability

Making the operation of the interlinker affordable is key in order to allow many parties to run it. We will estimate the cost of operation now.

Our transaction size ($\mathsf{txBytes}$) is constant at exactly 244 bytes. The median Bitcoin Cash fee per byte ($\mathsf{feePerByte}$) at the time of writing (November 2018) is 1 satoshi. Multiplied by our transaction size this gives us a transaction fee ($\mathsf{txFee} = \mathsf{txBytes} * \mathsf{feePerByte}$) of 244 satoshis. Our complete estimations based on the current price of Bitcoin Cash can be seen on Table 3.1.

We provide two implementations of an interlinker which both run in production. We'll now look at the pros and cons of each.

3.7 Python implementation with Bitcoin-ABC

Our first implementation is built on top of Bitcoin-ABC. Bitcoin-ABC is the reference implementation for Bitcoin Cash in C++. It is a fork of the original Bitcoin codebase (now Bitcoin Core), and it was the first ever implementation to support Bitcoin Cash. It has a very active community of developers and users. Due to its heritage from Bitcoin Core the code is very well written and tested, and any new feature for the Bitcoin Cash chain appears on Bitcoin-ABC first.

We run Bitcoin-ABC as a full node and interface with it using JSON-RPC. The interlinker is a Python module which knows (a) the location and credentials to connect with the full node and

(b) the velvet fork genesis block id. We use the python-bitcoinrpc library [11] for the JSON-RPC communication.

After the interlinker makes sure the full node is fully synced it starts waiting for new blocks. As described above we need our interlinker to get notified whenever there's a new block so that it can send a transaction with the new interlink for inclusion in the upcoming block. There's two options to get notified for new blocks from a full node.

3.7.1 Block Discovery

The first option is to utilize the ZeroMQ [12] functionality of Bitcoin-ABC. If compiled with the appropriate flags, Bitcoin-ABC can create a ZeroMQ channel where it will send notifications for new blocks and transactions. While the method seems very modern, it has some pitfalls. The main pitfall is race conditions: it is possible that the node pushes out a ZeroMQ notification about a block, however that block has not finished saving in the node's database, causing an immediate `getblock` request on the block to fail. Also, the ZeroMQ functionality is not enabled by default: one needs to compile the node with specific flags. While both issues are not fatal, in order to keep the interlinker as compatible with existing nodes as possible and to avoid workarounds we decided against using this functionality.

What we ended up using is busy polling through JSON-RPC with the `getbestblockhash` method. Every 5 seconds the interlinker will check the best block hash and if it doesn't match the one previously given then this means that there's been a new block.

```
1 tip_id = ''
2 while True:
3     possibly_new_tip_id = rpc.getbestblockhash()
4     if possibly_new_tip_id == tip_id:
5         sleep(NEW_TIP_CHECK_INTERVAL_SECONDS)
6         continue
7     tip_id = possibly_new_tip_id
```

3.7.2 Reaction to a New Block

When there is a new block, the interlinker computes the correct interlink for it, by updating the interlink with all the intermediate blocks from the velvet fork genesis up to and including the new block. It then computes the Merkle Tree root and includes it in an SPV tagged output. Subsequently it wraps the output inside a change transaction and sends it to the network.

We also see the use of `shelve` on the code below. `shelve` is a Python library which allows us to persist Python objects on disk and access them on demand. We will see promptly how we utilize `shelve` to act as a database of computed interlinks in order to speed up our computations.

```
1     logger.info('new block "%s"', tip_id)
2     with shelve.open(db_path) as db:
3         new_interlink = interlink(tip_id, db)
4     logger.debug('new interlink "%s"', new_interlink)
5     logger.info('mtr hash "%s"', new_interlink.hash().hex())
6     logger.info('velvet tx "%s"', send_velvet_tx(new_interlink.hash()))
```

3.7.3 Calculating the Interlink

The naive way to calculate the interlink would be starting from the velvet genesis block to keep a running interlink and sequentially update it. Considering that between the tip and the velvet genesis there could be millions of blocks, and for each block we need to issue a JSON-RPC call to get its block hash by using `getblockhash`. We experimentally found this method to take upwards of 15 seconds for approximately 15000 blocks, averaging at 1000 blocks/second.

To make this process faster we cache all interlinks we compute. This caching happens inside the `store` shown in the code below. `store` is a dictionary which maps block ids to their corresponding interlinks. In order to compute an interlink for a given tip we traverse the blockchain from the tip until we end up at a block for which the interlink is already available. We always ensure the cache contains at least the interlink for the velvet genesis which is empty and only contains the genesis pointer (of level ∞). We keep the block ids between the tip and the block with the cached

interlink and then proceed to update the interlink that was found with each intermediate block id, in order. In doing so we also save each interlink we compute to its corresponding block id in the store, so that it will be available if we ever need it again. After the interlink has been updated with all blocks we are done and we can return it as the final result.

```
def interlink(tip_id, store=None):
    store = {} if store is None else store
    intermediate_block_ids = deque()
    intermediate_id = tip_id
    if VELVET_FORK_GENESIS not in store:
        store[VELVET_FORK_GENESIS] = genesis_interlink()
    while intermediate_id not in store:
        intermediate_block_ids.appendleft(intermediate_id)
        intermediate_id = prev(intermediate_id)

    intermediate_interlink = store[intermediate_id]
    for block_id in intermediate_block_ids:
        intermediate_interlink = store[block_id] = \
               intermediate_interlink.update(block_id, level(block_id))
    return intermediate_interlink
```

3.7.4 Interlink

It's time to look at how we programmatically represent the interlink. We take the simplest approach which is to represent it as a POPO (Plain Old Python Object).

To instantiate an interlink one needs a velvet genesis id and optionally a list of blocks. This list of blocks represents the interlink pointers, where the element at $block[i]$ is the id of the most recent block at level i. If the list of blocks is not provided the interlink is initially empty.

```
class Interlink:
    def __init__(self, genesis, blocks=None):
        self.genesis = normalized_block_id(genesis)
        self.blocks = [] if blocks is None else blocks
```

The `update` method is a straightforward adaptation of Algorithm 4. It's important to note that instead of mutating the `self` object, `update` operates on a new copy which it then returns. This makes any `Interlink` object in essence immutable which allows us to cache it very elegantly.

```
    def update(self, block_id, level):
        block_id = normalized_block_id(block_id)
        blocks = self.blocks.copy()
        for i in range(0, level+1):
            if i < len(blocks):
                blocks[i] = block_id
            else:
                blocks.append(block_id)
        return Interlink(genesis=self.genesis, blocks=blocks)
```

The commitment is generated by the `hash` method defined on `Interlink`, which after getting the representation of the interlink as an array, including the velvet genesis as the last element, then calculates the Merkle Tree root (abbreviated as `mtr`) of a tree with the elements of this array as leaves.

```
    def as_array(self):
        return self.blocks + [self.genesis]

    def hash(self):
        return mtr(self.as_array())
```

3.7.5 Bitcoin-style Merkle Trees

For implementing the `mtr` function seen above we searched for existing Python libraries offering this functionality. While some libraries came up, none of them implemented the Bitcoin-style Merkle Trees we were after. This functionality is definitely implemented in some of more general Python Bitcoin libraries we make use of however it isn't exposed. This led us to write our own implementation of Bitcoin-style Merkle Trees.

We only implement the root calculation as described in Section 2.1.3 in `mtr`. `mtr` accepts an array of leafs. The function works by taking a level of the tree (initially the leafs) and repeatedly generating the upper level. This process continues until the level reached only has one element, where it's declared the root and is returned as the result. The function for generating the upper level, `next_level` uses many Python idioms and this requires an explanation. It starts by taking a level and then pairing its element with its adjacent. This happens with `zip_longest` which takes 2 lists and "zips" them together like so: `zip_longest([1, 2, 3], [4, 5, 6]) = [(1, 4), (2, 5), (3, 6)]`. For the uninitiated we provide a Haskell implementation below.

```
zip_longest :: [a] -> [a] -> a -> [a]
zip_longest (x:xs) (y:ys) fillvalue = (x, y) : zip_longest xs ys fillvalue
zip_longest (x:xs) [] fillvalue = (x, fillvalue) : zip_longest xs [] fillvalue
zip_longest [] (y:ys) fillvalue = (fillvalue, y) : zip_longest [] ys fillvalue
zip_longest [] [] _ = []
```

The two lists passed to `zip_longest` are `level[::2]` which is all even elements of `level` and `level[1::2]` which is all odd elements. We use `zip_longest` instead of the regular `zip` to account for the case where `level` is of odd length and we need to hash the last element with itself.

```
    from itertools import zip_longest
    def next_level(level):
        return [hash_siblings(l, r) for l, r in zip_longest(level[::2], level[1::2],
             fillvalue=level[-1])]
```

After getting the list of pairs we go through it hashing the pairs together.

```
    from hashlib import sha256
    def hash_siblings(l, r):
        h1, h2 = sha256(), sha256()
        h1.update(l)
        h1.update(r)
        h2.update(h1.digest())
        return h2.digest()
```

This setup makes our Merkle Tree root calculation very straightforward.

```
def mtr(leafs):
    assert len(leafs) > 0
    level = leafs
    while len(level) > 1:
        level = next_level(level)
    return level[0]
```

3.7.6 Velvet Transactions

For creating the SPV tagged outputs we utilize the python-bitcoinlib library [31]. We construct the outputs array including a single script with the `OP_RETURN`, our SPV tag which is the ASCII encoding for `interlink` and our payload which we expect to be our interlink commitment. We then serialize a transaction with this output.

```
from bitcoin.core import CMutableTxOut, CScript, CMutableTransaction, OP_RETURN
def create_raw_velvet_tx(payload_buf):
    VELVET_FORK_MARKER = b'interlink'
    digest_outs = [CMutableTxOut(0, CScript([OP_RETURN, VELVET_FORK_MARKER, payload_buf]))]
    tx = CMutableTransaction([], digest_outs)
    return tx.serialize().hex()
```

The next stage is to actually send the transaction. Of course a transaction with only one output is incomplete and invalid. In order to fill it in with the appropriate inputs and change output we turn to our Bitcoin-ABC node. We use the `fundrawtransaction` JSON-RPC method in order to do all the above. We then only have to sign and send the transaction out in the network.

```
def send_velvet_tx(payload_buf):
    change_address = rpc.getaccountaddress("")
    funded_raw_tx = rpc.fundrawtransaction(create_raw_velvet_tx(payload_buf),
           {'changeAddress': change_address})['hex']
    signed_funded_raw_tx = rpc.signrawtransaction(funded_raw_tx)['hex']
    return rpc.sendrawtransaction(signed_funded_raw_tx)
```

3.7.7 Funding

It is important to note here that in order to send the transactions, the interlinker has to pay transaction fees as seen earlier. However, we haven't talked about the interlinker controlling a wallet or private keys which would make it seem like there is no way to fund the transactions. Because of how the JSON-RPC methods work we don't need to specify a private key or wallet external to the full node and we can let the full node create and pay for our transactions using its default wallet. Thus the way to fund the interlinker is to fund the default wallet of the full node.

3.7.8 Dependency Management

We utilize Pipenv [27] for dependency management. Pipenv makes it easy to have a centralized file with a list of all dependencies, as well as a so-called lock file where each dependency is absolutely specified with a specific version and hash for its wheel file. It is great for reproducing the exact environment where the application is well-tested. It also makes installation of all dependencies very easy with just `pipenv install`.

3.7.9 Testing

The code is thoroughly unit-tested using pytest [20]. The Interlink implementation is a very crucial part of our implementation and is tested as follows.

```
from interlink import Interlink

def test_interlink():
    interlink = Interlink(genesis=b'\x01')
    interlink = interlink.update('0c', 5)
    interlink = interlink.update('0b', 4)
    interlink = interlink.update('0a', 2)
    assert interlink.as_array() == [b'\x0a', b'\x0a', b'\x0a', b'\x0b', b'\x0b', b'\x0c', b'\x01']

def test_interlink_cascading():
    interlink = Interlink(genesis=b'\x02')
    interlink = interlink.update('0a', 2)
    interlink = interlink.update('0b', 4)
    interlink = interlink.update('0c', 5)
    assert interlink.as_array() == [b'\x0c'] * 6 + [b'\x02']

def test_interlink_str():
    assert str(Interlink(genesis=b'\x02')) == '[1 * 02]'
```

The Bitcoin-style Merkle Trees are also very important to get right. To this end we generate testcases adapted from actual Bitcoin blocks by making sure the merkle tree root of their transaction ids matches the one found in the block. We also make sure to also test the boundaries (for example having 0 and 1 leafs).

```
import pytest
import json

from merkle import mtr

def do_test_fixture(fixture_path):
    with open(fixture_path, 'r') as f:
        fixture = json.load(f)
    expected_merkleroot = bytes.fromhex(fixture['merkleroot'])[::-1]
    txs = [bytes.fromhex(tx_id)[::-1] for tx_id in fixture['txs']]
    assert mtr(txs) == expected_merkleroot

def test_bitcoin_block_100():
    do_test_fixture('./fixtures/merkle/bitcoin_block_100.json')

def test_bitcoin_cash_testnet_2_txs():
    do_test_fixture('./fixtures/merkle/bitcoin_cash_testnet_2_txs.json')

def test_bitcoin_cash_testnet_big():
    do_test_fixture('./fixtures/merkle/bitcoin_cash_testnet_1259110.json')

```

```
def test_no_leafs():
    with pytest.raises(AssertionError):
        mtr([])
```

3.7.10 Distribution

Our software is production-grade and we release it under the open source MIT license. [2].

3.8 JavaScript implementation with bcash

The second implementation is build on top of the bcash JavaScript library. bcash implements a full node and wallet functionality exclusively in JavaScript without being based on the Bitcoin C++ codebase as did most previous solutions. A major advantage of bcash is that it implements an SPV node too, which is very useful since the interlinker only needs the block ids of the chain but doesn't need to know about the block contents, so requiring a full node to run it would be a waste of space and bandwidth.

3.8.1 Architecture

The bcash architecture is a departure from the JSON-RPC paradigm described above and requires some explanation. One major difference is that this implementation doesn't require an external node: the interlinker is a standalone program which due to the bcash library is also an SPV node. bcash offers Node classes which are standalone nodes. Apart from SPV it also offers a `FullNode` class.

Let's focus on `SPVNode` for now. The node is mainly composed of the following subcomponents:

- A `chain` object which holds the current blockchain.
- A `pool` object which is used for bidirectional communication with other nodes in the network.
- A `mempool` object which holds the unconfirmed transactions in the network.

These components are the ones crucial for the node to function. bcash offers more functionality for nodes such as JSON-RPC and HTTP APIs, which we will not make use of.

bcash is designed to be very modular so not much logic is found within its `Node` classes, as they only wire the components they are composed of together. The architecture is ripe for such wiring as bcash makes heavy use of the event system JavaScript is known for.

What we implement is an `InterlinkerNode` class which augments the functionality of `SPVNode` by extending it.

3.8.2 Block Discovery

bcash makes it very easy to know when a new block has been discovered: its `SPVNode` class offers a `block` event that we can listen on for new blocks.

```
  this.on('block', async (blk) => {
    this.ourLogger.info('got block event (blkid=%s, height=%d)', blk.rhash(), blk.height);
    await this.doInterlink();
  });
```

3.8.3 Synchronization

Sending velvet transactions while we still are not completely synced is not desired behavior. Unfortunately, bcash does not provide any mechanism in order to know when the Initial Block Download is complete.

We work around this by use of debounce. First we should explain what debounce is: when we debounce a function `foo` we should also provide a delay in milliseconds. For example we could do

[2]The code is available at https://github.com/decrypto-org/bch-interlinker

`bar = debounce(foo, 1000)` where `bar` is a new function. We can then call `bar` with the exact same arguments we would `foo`, however debouncing makes it so that `foo` will only be called after 1000 milliseconds have passed since the last call to `bar`. This technique is usually utilized to avoid having a user double-click a button perform a destructive action twice. It's useful here because we can have a function `onSync` that we debounce and now call `maybeSync`. We attach `maybeSync` to the `block` event we saw earlier. We assume that when our node is not synced we can expect multiple subsequent `block` events. We guard against these events because of the debounce. When we're satisfied that enough time with no events occurring has passed, finally `onSync` is actually called, which removes the `maybeSync` handler and sets the final handler to interlink on each block that we saw previously.

It is important to note that this is only a heuristic but it has proven to work well enough during our testing.

```
onSync() {
  this.ourLogger.info('chain synced');
  this.wallet = this.walletDB.primary;
  this.removeListener('block', this.maybeSynced);
  this.on('block', async (blk) => {
    this.ourLogger.info('got block event (blkid=%s, height=%d)', blk.rhash(), blk.height);
    await this.doInterlink();
  });
  this.doInterlink().then();
}
```

3.8.4 Reaction to a New Block

Whenever a new block occurs we calculate the interlink and send a transaction with its commitment just as before. This code looks almost identical in form to the previous implementation: we compute the interlink, calculate the commitment, create a transaction with a tagged SPV output and send it. Here `wallet.send` additionally takes care of the funding we saw in the previous implementation.

```
async doInterlink() {
  const interlink = await this.getInterlinkSinceBlock(VELVET_GENESIS);
  const hash = interlink.hash();

  this.ourLogger.info('interlink hash = %s', hash.toString('hex'));
  try {
    const tx = await this.wallet.send({
      outputs: [bcash.Output.fromScript(taggedSPVOutput(hash), 0)],
    });
    this.ourLogger.info('sent tx (txid=%s)', revHex(tx.hash()));
  } catch (e) {
    this.ourLogger.error(e);
    this.ourLogger.error('not enough funds to publish interlink tx');
    this.ourLogger.error('feed me: %s', await this.wallet.receiveAddress());
    return;
  }
}
```

3.8.5 Calculating the Interlink

Here we implement the naive approach that we discussed in the previous implementation for calculating the interlink. We can safely do this as there are no speed concerns, seeing that we have direct access to our chain and no JSON-RPC calls need to transpire here. This makes this procedure almost instant.

```
async getInterlinkSinceBlock(blockId) {
  const interlink = new Interlink();
  let blk = await this.chain.getEntryByHash(blockId);
  let blockCount = 0;
  let lastBlock = blk;
  while (blk) {
    ++blockCount;
    lastBlock = blk;
    interlink.update(blk.hash);
```

```
      blk = await this.chain.getNextEntry(blk);
    }
    this.ourLogger.info('saw %d blocks', blockCount);
    this.ourLogger.info('last block (id=%s, height=%d)', revHex(lastBlock.hash), lastBlock.height);
    return interlink;
  }
```

3.8.6 Interlink Representation

In similar fashion to the previous implementation we represent the interlink as a Plain Old JavaScript Object. Note that here our `update` is destructive in contrast as we don't utilize the immutability for caching.

```
class Interlink {
  constructor() {
    this.list = [];
  }

  update(blockId) {
    const lvl = level(blockId, MAX_TARGET);
    for (let i = 0; i <= lvl; ++i) {
      if (i < this.list.length)
        this.list[i] = blockId;
      else
        this.list.push(blockId);
    }
  }
}
```

Calculating the commitment here is much easier here, as bcash provides as with a good API for creating and manipulating Bitcoin-style Merkle Trees.

```
  hash() {
    return merkle.createRoot(hash256, [...this.list])[0];
  }
```

3.8.7 Velvet Transactions

We create SPV tagged outputs by leveraging the Script API offered by bcash. All we need to do is simply create an empty script and then push in sequence an `OP_RETURN`, our SPV tag and then the payload.

```
const {Script, MTX} = require('bcash');

const SPV_TAG = Buffer.from('interlink');

module.exports = function taggedSPVOutput(buffer) {
  const script = new Script();
  script.pushOp(Script.opcodes.OP_RETURN);
  script.pushData(SPV_TAG);
  script.pushData(buffer);
  return script.compile();
};
```

3.8.8 Distribution

Our software is production-grade and we release it under the open source MIT license. [3].

3.9 Deployment

At the time of writing of this thesis, both implementations run in production on the Bitcoin Cash testnet chain and have been running for over 3 months. We started with our first deployment late September 2018, with the first prototype of our Python interlinker. This gave us a chance to

[3]The code is available at https://github.com/gtklocker/bcash-interlinker-js

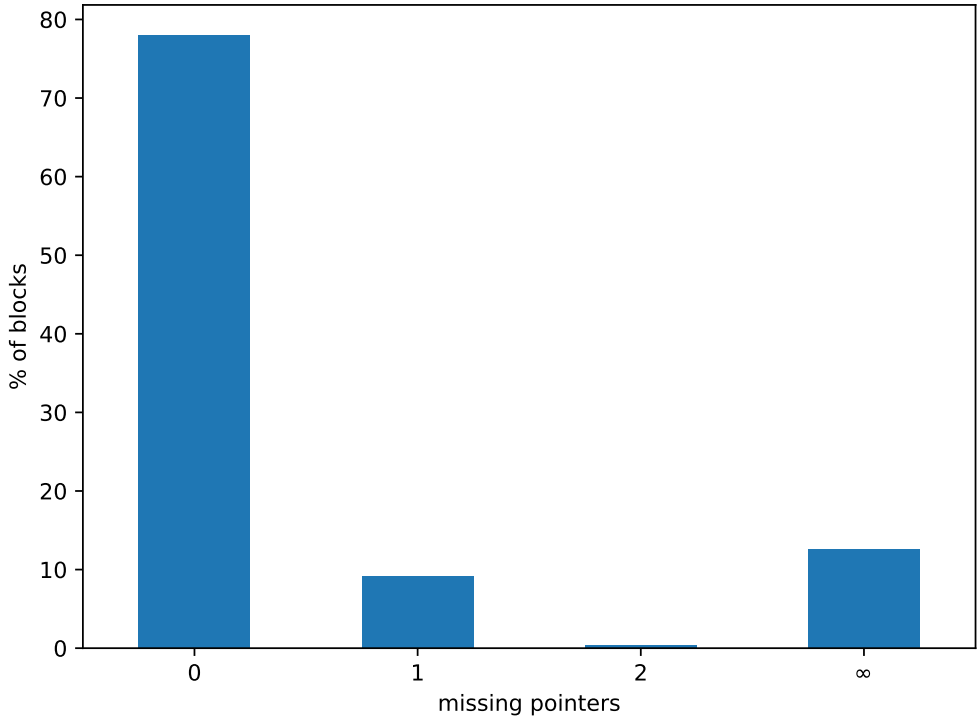

Figure 3.4: Reliability of our testnet deployment.

catch bugs that only appear on long running processes and make sure our software is resilient to network and other library failures. As new features and bug fixes kept rolling in our software we continuously updated the deployments.

3.9.1 Docker

In order to minimize the dependencies and complexity of running this software we rely on the industry standard, Docker [23]. With Docker we specify the steps required to build an image with all required dependencies that acts like an executable. Docker also allows our service to run in an isolated sandbox without access to the rest of the host system.

We provide `Dockerfile`s for both our implementations.

3.9.2 docker-compose

We also rely on docker-compose [14] in order to make deployments easy, having a single configuration which describes for example how the ABC node and our Python interlinker should be wired together in order to work well. All in all, this makes for a very quick and painless deployment process, allowing any interested party to run the interlinker on their own. docker-compose was especially invaluable in painlessly keeping our deployment in sync with our continuously changing codebases.

We provide reference `docker-compose.yml` files for both our implementations.

3.10 Experimental Data

We will now examine real-world data gathered from our Bitcoin Cash testnet deployment. On Figure 3.4 we can look at the reliability of our interlinker deployment. We can see that about 80% of the blocks have been correctly interlinked with our velvet transactions, and about 10% of them only have 1 interlink pointer missing, which makes them potentially usable. With ∞ we denote the

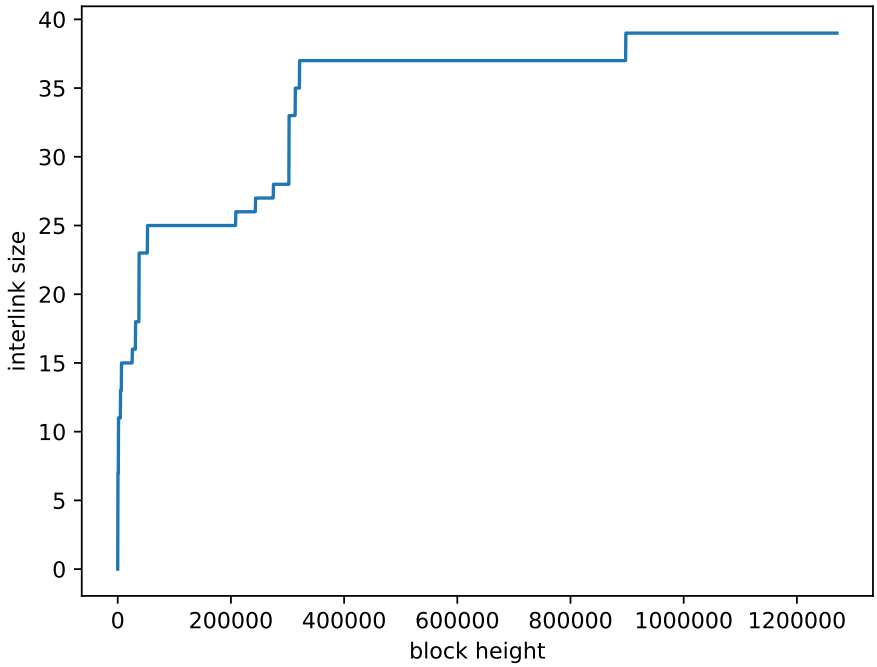

Figure 3.5: Level distribution on Bitcoin Cash testnet.

case where all pointers are missing due to a block only having incorrect interlinks or no interlinks at all.

In order to estimate how our deployment would perform in time we can assume that our velvet genesis matches the real genesis. On Figure 3.5 we see the growth of the interlink size on the Bitcoin Cash testnet. We also look at the interlink sizes on Bitcoin and Bitcoin Cash mainnet on Figure 3.6, which is a more realistic environment, where we notice that the level increase is less steep than on testnet.

3.10.1 Methodology

For our estimations it was required that we obtained the corresponding chains. However, obtaining the full chains as we have seen is cumbersome, as hundreds of gigabytes of data need to be downloaded for every chain. We noticed that for our purposes only the headers were required, but even with this observation there was still no easy way to obtain the header chain for major cryptocurrencies. To this end we implemented a utility in Haskell called `blockheaders-dl` [4]. The utility connects to Electrum servers corresponding to the requested cryptocurrency and repeatedly queries the headers until all the headers are downloaded. Bitcoin, Bitcoin Cash and Litecoin are currently supported. With this utility we were able to download and process all these blockchains in a few minutes, whereas synchronizing full nodes for each would take more than a week.

3.11 Interlink Compression

We will now revisit the interlink encoding. Recall that we created the interlink encoding by creating a Merkle Tree with the interlink elements as its leafs and taking its root. We call this the *block list* construction. During our deployment on the Bitcoin Cash network it became evident that an interlink block list construction contained lots of repetition. This repetition is not a problem in the interlink encoding itself, as it is constant. However if for example half the elements of a block

[4]The code is available at https://github.com/gtklocker/blockheaders-dl

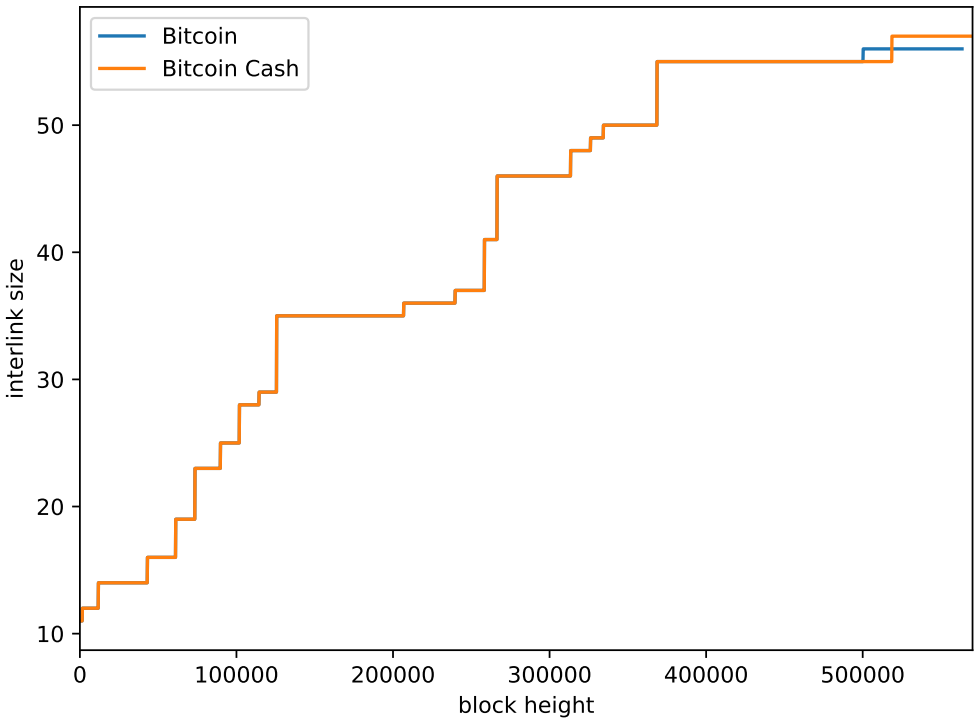

Figure 3.6: Size of the interlink assuming the real genesis on Bitcoin and Bitcoin Cash mainnet.

list are duplicates then the Merkle Tree would be one level higher and any inclusion proof would take up one more item than if the leafs did not contain duplicates.

We call our construction a *block set* construction, because each block id is included only once in the Merkle Tree leaves. We evaluated our construction on the Bitcoin Cash mainnet, assuming that our velvet fork genesis is the actual genesis at height 0 (unlike our velvet genesis). This allows us to examine how our solution would behave in time. In Figure 3.7 we see a comparison between the size of an interlink inclusion proof using both constructions. The block list inclusion proofs appear to grow with the size of the chain, whereas the block set inclusion proof remain almost constant around 3. The relative savings are shown in Figure 3.8, where as the chain grows we see savings of up to 50%.

An example of how a block list can end up containing many duplicates is not difficult to imagine. Suppose a very high-level μ-superblock is mined, and that for some period after that, at best $(\mu - \lambda)$-superblocks are mined. For that period, the block list will contain λ duplicates.

It is important that the leafs in the block set are consistently ordered, so that it can be easy for provers to verify the included commitment is correct. An option is ordering by value, where the leafs are in ascending or descending block ID order. Another option is ordering by level, where the block IDs are in order from level 0 to ∞ as usual, but duplicates are eliminated. For both cases, the genesis id is always assumed to be included at a specific location, either at the beginning or the end.

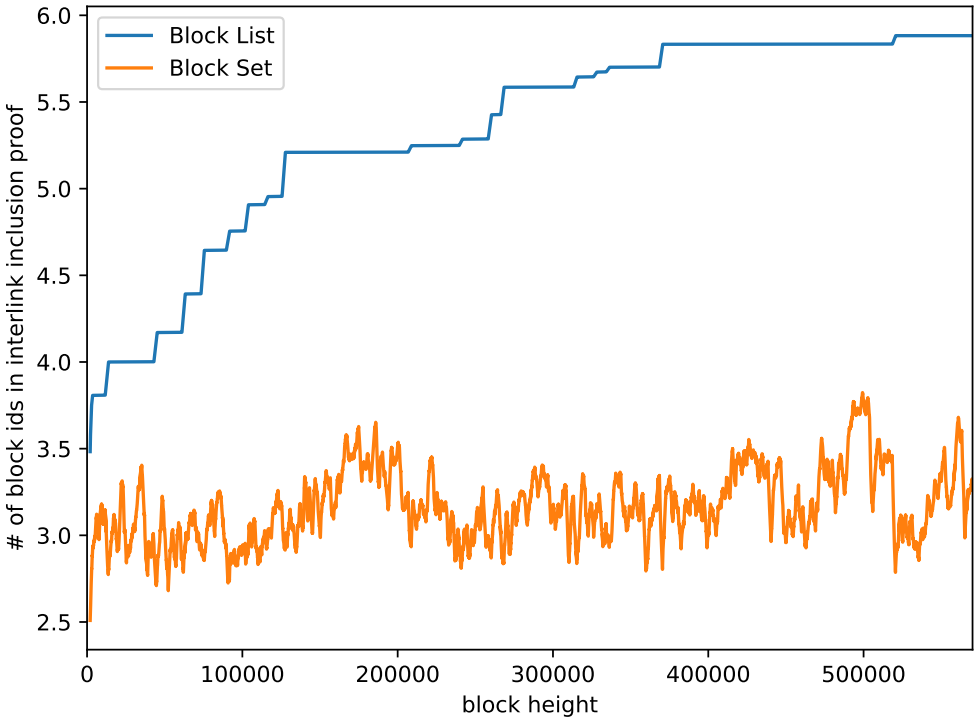

Figure 3.7: Number of elements on an interlink inclusion proof using the block list and block set constructions.

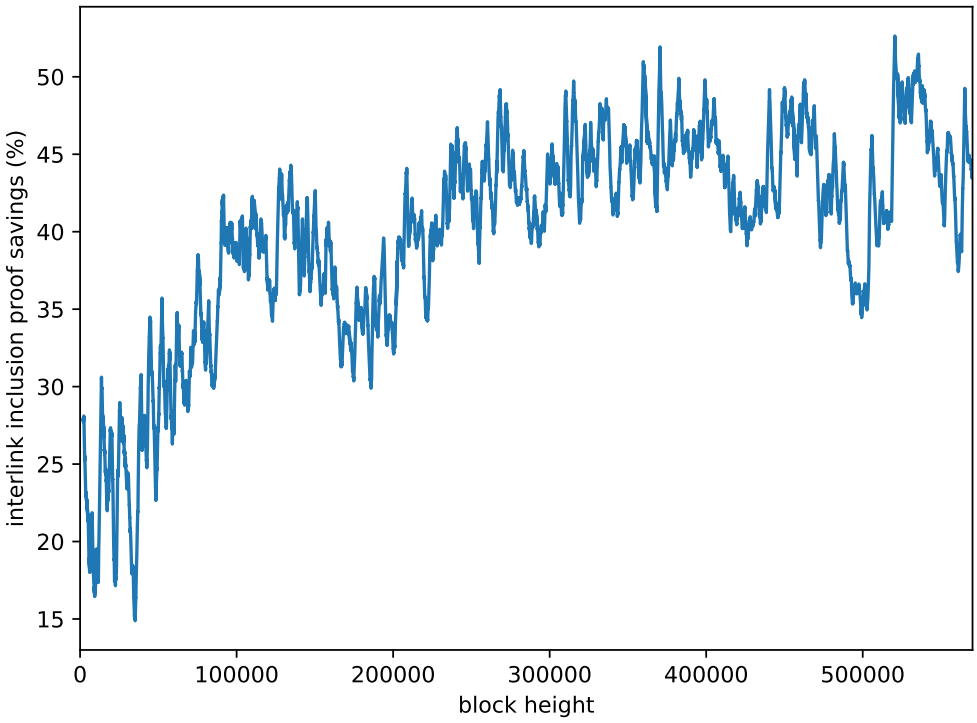

Figure 3.8: Interlink inclusion proof savings by utilizing the block set construction.

Chapter 4

NIPoPoW Velvet Fork Prover

Now that we've established a fork on the Bitcoin Cash chain, we show how our fork can be put to use by creating proofs. We introduce a kind of client called a *prover* which can create all kinds of NIPoPoW proofs on demand.

Let's assume we have an SPV node. We set the bloom filter to our velvet fork tag in order to get all blocks containing only the transactions of the fork. It's possible to then extract from each transaction the payload, which should be the interlink commitment.

bcash, also used for the interlinker earlier, turns out the be the only option for an SPV node on Bitcoin Cash, making it the obvious choice for the prover.

4.1 Overview

Our prover works as follows. We expect a user to run the prover, passing as a parameter if they want a suffix or infix proof. If they desire an infix proof they have to provide the block of interest. Once they do that the prover runs and syncs with the Bitcoin Cash blockchain via SPV. It does some similar work to the work that an interlinker does: it knows what the correct interlink is for each block. It then checks the included commitments on each block and keeps a valid commitment for each block if found. A block containing a valid commitment is called a *valid velvet*. Interlink pointers can only be used on valid velvets. Once the chain is synced and the prover knows the valid velvets along with the whole blockchain it creates a proof of the requested type and outputs it in JSON format.

4.2 Interlink Implementation

Implementing an interlink structure is something we've already covered many times. Our implementation here is based on immutable value objects holding the interlink. Updating an interlink is done in the usual manner and creates a new interlink. What's different here is that we have a `proof` method: we use this method to generate a proof of inclusion of a specific pointer in the interlink commitment. We also need a way to be able to get a pointer from a specific level on the interlink and this is accomplished with the `at` method. Note that `at` can be asked for a level higher than the size of the interlink, in which case the velvet genesis is returned as it's assumed to be a block of level ∞.

```
1  class Interlink {
2    list: Array<BlockId>;
3    genesisId: BlockId;
4
5    constructor(genesisId: BlockId, list: Array<BlockId> = []) {
6      this.genesisId = genesisId;
7      this.list = list;
8    }
9
10   update(blockId: BlockId) {
11     const list = this.list.slice();
12     const lvl = level(blockId);
```

```
    for (let i = 0; i <= lvl; ++i) {
      if (i < list.length) list[i] = blockId;
      else list.push(blockId);
    }
    return new Interlink(this.genesisId, list);
  }

  proof(level: number) {
    return merkle.createBranch(hash256, level, [...this.list]);
  }

  hash() {
    return merkle.createRoot(hash256, [...this.list])[0];
  }

  get length() {
    return this.list.length;
  }

  at(lvl: number) {
    if (lvl >= this.list.length) {
      return this.genesisId;
    }
    return this.list[lvl];
  }
}
```

4.3 Locating Velvet Transactions

Locating the velvet transactions is very easy considering that all our velvet transactions are SPV tagged. The only thing we have to do is add the SPV tag in our bloom filter. We do this right before connecting to the network.

```
module.exports = class ProverNode extends bcash.SPVNode {
  // ...
  async connect() {
    this.pool.watch(Buffer.from(VELVET_FORK_MARKER));
    await super.connect();
  }
  // ...
}
```

4.4 Extracting Interlink Commitments

we have to check that the transaction the valid form described earlier (`OP_RETURN <tag> <commitment>`). In order to do that we look at the outputs of each transaction and only keep any outputs of our form. If some output matches then we extract the script's third element which is the commitment.

```
const extractInterlinkHashes = compose(
  map(
    compose(
      head,
      drop(1)
    )
  ),
  filter(isInterlinkData),
  map(
    compose(
      getCleanScriptData,
      prop("script")
    )
  ),
  filter(isBurnOutput),
  prop("outputs")
);
```

4.5 Handling Merkle Blocks

When a block or transaction of a block matches our bloom filter, the block is relayed by full nodes as a `merkleblock` message. MerkleBlocks contain the header of the actual block as well as the transactions ids that matched. It also contains a proof of inclusion for all the transactions ids provided. One can check this proof of inclusion with accordance to the `merkleRoot` inside the block header. The transactions are included in subsequent `tx` messages. bcash hides this process from the user: on a new block event a MerkleBlock is provided and it contains a `txs` record, where the transactions that matched are included in full.

In order to check if we have matched transactions in a block all we have to do is check if its `.txs` array is non-empty. If it is not then this block is definitely not a valid velvet. If it is not, then we have to ensure a correct form and extract the commitment, also saving it. We utilize the method `extractInterlinkHashes` and map it over all transactions in the block.

```
const extractInterlinkHashesFromMerkleBlock = compose(
  unnest,
  map(extractInterlinkHashes),
  prop("txs")
);
```

4.6 RealLink & Verifying Interlink Commitments

We extend the `realLink` discussed earlier. Other than allowing us to get the interlink of any block it also allows us to check if a block is a valid velvet. It is able to do that as it gets notified for every new block seen during the synchronisation. When a new block appears, it gets informed about the block id and the interlinks included in the block. It keeps a running interlink in the same fashion the interlinker does so because the blocks events happen in-order it can check whether its running interlink matches the ones included in the block. In case there's a match the block is marked as a valid velvet and the running interlink is updated so that RealLink can be ready to process the next block.

```
module.exports = class RealLink {
  blockIdToInterlink: BufferMap;
  runningInterlink: Interlink;
  validBlocks: BufferSet;

  constructor(genesisId: BlockId) {
    this.blockIdToInterlink = new BufferMap();
    this.runningInterlink = new Interlink(genesisId);
    this.validBlocks = new BufferSet();
  }

  onBlock(newBlockId: BlockId, interlinks: Array<Buffer>) {
    if (this.blockIdToInterlink.has(newBlockId)) {
      return;
    }

    this.blockIdToInterlink.set(newBlockId, this.runningInterlink);
    if (
      interlinks.some(interlink =>
        interlink.equals(this.runningInterlink.hash())
      )
    ) {
      this.validBlocks.add(newBlockId);
    }
    this.runningInterlink = this.runningInterlink.update(newBlockId);
  }

  get(blockId: BlockId) {
    return this.blockIdToInterlink.get(blockId);
  }

  hasValidInterlink(blockId: BlockId) {
    return this.validBlocks.has(blockId);
  }
```

```
};
```

4.7 Handling Merkle Blocks Redux

All information regarding the chain is sorted in a `VelvetChain`. A `VelvetChain` is the main entrypoint for block events. It is the single source of truth about the blockchain. When there is a new block, the whole block is associated with its id for later lookup, and its height is also recorded. It then gathers the block's interlink commitments and forwards them as a new event to `realLink`.

```
// ...
onBlock(blk: bcash.MerkleBlock, height: number) {
  const id = blk.hash();
  if (this.blockById.has(id)) {
    return;
  }

  ++this.height;

  if (!this.genesis) {
    this.genesis = id;
    this._realLink = new RealLink(this.genesis);
    console.log("genesis was %O", blk);
  }

  this.blockById.set(id, blk);
  this.heightById.set(id, this.height);
  this.blockList.push(blk.hash());

  const includedInterlinkHashes = extractInterlinkHashesFromMerkleBlock(blk);
  this.realLink.onBlock(id, includedInterlinkHashes);

  this.lastBlock = id;
}
// ...
```

4.8 Following Up

We implement the `followUp` algorithm as described in Algorithm 8. `followUp` acts on a `VelvetChain` and requires a `newerBlockId` which is equivalent to B and μ.

We also see use of the `levelledPrev` function. `levelledPrev`, given a block id and a level will attempt to cross the interlink pointer at the requested level. If the block is not a velvet block it will fail and just return the `previd`.

A departure from the exact algorithm is that in our implementation we don't return a block set. Instead we return a block list which is sorted by height. By convention we also say that a block A is *left* of another block B if $B.\mathsf{height} > A.\mathsf{height}$.

```
followUp(newerBlockId: BlockId, mu: Level): Array<BlockId> {
  const genesis = nullthrows(this.genesis);
  let id = newerBlockId;
  let path = [id];
  while (!id.equals(genesis)) {
    id = this.levelledPrev(id, mu);
    path.push(id);

    if (level(id) === mu) {
      break;
    }
  }

  path.reverse();
  return path;
}
```

4.9 Finding the Velvet Upchain

Utilizing the `followUp` algorithm to implement Algorithm 9 is also pretty straightforward. However instead of only $\mathcal{C}\uparrow^{\mu}$ we would like to implement $\mathcal{C}\uparrow^{\mu}$ $\{B :\}$ as this is needed by the prover. Programmatically we reference B by its id as `leftBlockId`. The original algorithm repeatedly calls to `followUp` until it has reached the genesis, which in code can be optimized away as we know that we only need blocks after (and including) B. This is why we `followUp` until B only, and take care of the special case where B is not a μ superblock. We look for the path from `followUp` for any blocks which are left of B in the chain. If that is the case, the algorithm only keeps the part of the path which is right of B, and B if it is included in the path and then exits.

```
findVelvetUpchain(
  mu: Level,
  leftBlockId: BlockId,
  rightBlockId: ?BlockId = this.lastBlock
): {
  muSubchain: Array<BlockId>,
  wholePath: Array<BlockId>
} {
  const genesis = nullthrows(this.genesis);
  let id = nullthrows(rightBlockId);

  let wholePath = [];
  let muSubchain = [];
  if (level(id) >= mu) {
    muSubchain.push(id);
  }

  let goneTooFar = false;
  while (!id.equals(leftBlockId) && !id.equals(genesis) && !goneTooFar) {
    let path = this.followUp(id, mu);
    let outOfRangePathIndex = _.findLastIndex(path, x =>
      this.isLR(x, leftBlockId)
    );
    if (outOfRangePathIndex !== -1) {
      path.splice(0, outOfRangePathIndex + 1);
      goneTooFar = true;
    }

    id = path[0];

    if (level(id) >= mu) {
      muSubchain.push(id);
    }
    wholePath = path.slice(1).concat(wholePath);
  }

  wholePath = [id, ...wholePath];

  muSubchain.reverse();
  return {
    muSubchain,
    wholePath
  };
}
```

4.10 Crafting Suffix Proofs

Using these primitives we can now move on to creating our first proofs. The interface implemented allows us to directly translate the pseudocode shown in Algorithm **??**. The code will first initialize the two parts of the proof, `pi` and `chi` to be empty. Then it will pick $max\mu$ to be the level of the rightmost stable block, $\mathcal{C}[-k]$, with B initially set to the genesis block, G. Then for each level from $max\mu$ up to and including 0, it utilizes `findVelvetUpchain` to get the velvet upchain from the rightmost stable block up to and including B. If the μ superblocks of the chain are more than m, the path returned is included and B is set to the m-th rightmost μ superblock of the path.

```
function suffixProof({
  chain: C,
  k,
  m
}: {
  chain: VelvetChain,
  k: number,
  m: number
}): Array<BlockId> {
  let leftId = nullthrows(C.genesis);
  let pi: Array<BlockId> = [],
    chi: Array<BlockId> = [];
  let rightMostStableId = C.idAt(-k - 1);
  let maxMu = C.interlinkSizeOf(rightMostStableId);

  for (let mu = maxMu; mu >= 0; --mu) {
    let { muSubchain, wholePath } = C.findVelvetUpchain(
      mu,
      leftId,
      rightMostStableId
    );
    let newBlocks = wholePath;
    if (mu > 0) {
      if (muSubchain.length <= m) {
        continue;
      }
      leftId = muSubchain[muSubchain.length - m];
      let leftIdInWholePath = newBlocks.findIndex(id => id.equals(leftId));
      newBlocks = newBlocks.slice(0, leftIdInWholePath);
    }
    pi = pi.concat(newBlocks);
  }

  for (let i = -k; i < 0; ++i) {
    chi.push(C.idAt(i));
  }

  return pi.concat(chi);
}
```

4.11 Following Down

As we noted in Section 2.4.7, a proof is a valid infix proof with regards to a set of blocks if and only if it contains those blocks, and it is traversable. This means th a suffix proof is also an infix proof for any possible subset of its included blocks (assuming the infix predicate requires only block header information which is included for every block).

If however, an infix proof for a specific block is required, it should be possible to augment our suffix proof with it. To do this, we implement `followDown`, as seen in Algorithm 5.

The purpose of the algorithm is to link two blocks, starting from the higher-level block on the right, `hi` and ending at the lower-level block on the left, `lo`. It attempts to take the largest possible steps, using the maximum-level pointer available on `hi`'s valid interlink which does not surpass the block of interest. If the interlink on some block the algorithm ends up is invalid, the block's prevId is used instead. Continuing in this manner, a traversable path is formed between `hi` and `lo`.

```
function followDown(C: VelvetChain, hi: BlockId, lo: BlockId) {
  let B = hi;
  let aux = [];
  let mu = level(hi);
  assert(C.heightOf(hi) >= C.heightOf(lo));
  while (!B.equals(lo)) {
    let Bp = C.levelledPrev(B, mu);
    if (C.heightOf(Bp) < C.heightOf(lo)) {
      --mu;
    } else {
      if (!B.equals(hi)) {
        aux = [B, ...aux];
      }
```

```
      B = Bp;
    }
  }
  return aux;
}
```

Following down is not enough to promote an existing suffix proof to a traversable infix proof as we will see now.

4.12 Going Back

Something the original NIPoPoWs paper doesn't address is that our block of interest may have an invalid interlink, thus it is not guaranteed that followDown will be enough. To mitigate this, we make sure to create a valid path of valid interlinks back to our suffix proof block, with a procedure called goBack, seen in Algorithm 10, which is very works similarly to followDown, but instead of attempting to downgrade to lower levels it attempts to upgrade to higher levels.

Algorithm 10 The goBack function which produces the necessary blocks to connect a block right to a preceeding block left.

```
1: function goBack(left, right, height, realLink, blockById)
2:     aux ← ∅
3:     while right ≠ left do
4:         for μ = level(left) down to 0 do
5:             if realLink[id(right)][μ] = right.interlink[μ] then
6:                 B ← blockById[right.interlink[μ]]
7:                 if height[B] ≥ height[left] then
8:                     aux ← aux ∪ {B}
9:                     right ← B
10:                    break
11:                end if
12:            end if
13:        end for
14:    end while
15:    return aux
16: end function
```

More specifically, suppose we have a suffix proof and we select two blocks, A and C. These blocks are selected so that they are adjacent in the proof and that B, our block of interest, is contained between them in the underlying blockchain. Also, C should be right of A to avoid ambiguity. By following down we obtain a traversable path from C to B. If we augment a suffix proof with this path, the result may not be traversable, as we have no guarantee that there is a valid path from B to A. In order for our final proof to be traversable we have to ensure a valid path between those two blocks.

To this end we implement `goBack`, which mimics `followDown` and connects `lo` to `hi` in the same manner.

```
function goBack(C: VelvetChain, lo: BlockId, hi: BlockId) {
  let aux = [];
  assert(C.heightOf(hi) <= C.heightOf(lo));
  while (!lo.equals(hi)) {
    for (let mu = level(hi); mu >= 0; --mu) {
      const b = C.levelledPrev(lo, mu);
      if (C.heightOf(b) >= C.heightOf(hi)) {
        lo = b;
        aux.unshift(lo);
        break;
      }
    }
  }
  aux.shift();
```

```
  return aux;
}
```

4.13 Crafting Infix Proofs

Using these tools we are able to craft our infix proofs. The implementation works only for a single block of interest, B, but can be easily extended to multiple such blocks. It is assumed that the infix predicate depends only on block header data. The algorithm starts by creating a standard suffix proof. In case the suffix proof already contains the block of interest the suffix proof is enough and can be returned. Otherwise, it will find the blocks in the proof before and after B: A and C. Then the proof returned will be a concatenation of the following:

- The suffix proof up to and including A.
- The path obtained via `goBack`, ordered from A to B.
- B.
- The path obtained via `followDown`, ordered from B to C.
- The rest of the suffix proof starting from C.

```
function infixProof({
  chain: C,
  blockOfInterest: B,
  k,
  m
}: {
  chain: VelvetChain,
  blockOfInterest: BlockId,
  k: number,
  m: number
}): Array<BlockId> {
  const suffixP = suffixProof({ chain: C, k, m });
  if (suffixP.some(x => x.equals(B))) {
    return suffixP;
  }

  const pi = suffixP.slice(0, -k),
    chi = suffixP.slice(-k);

  const afterBIndex = _.findIndex(pi, e => C.heightOf(e) >= C.heightOf(B));
  const afterBBlock = pi[afterBIndex];
  const beforeBIndex = Math.max(afterBIndex - 1, 0);
  const beforeBBlock = pi[beforeBIndex];

  return [
    ...pi.slice(0, afterBIndex),
    ...goBack(C, B, beforeBBlock),
    B,
    ...followDown(C, afterBBlock, B),
    ...pi.slice(afterBIndex),
    ...chi
  ];
}
```

4.14 Type Safety

To ensure code quality we opted to use Flow [15]. Flow allowed us to have type safety while keeping all the good characteristics of JavaScript. Unfortunately this introduced a couple of complications.

One is that our source files are not valid JavaScript anymore. In order to run our code we need to pass it through a pre-processor like Babel which emits clean JavaScript that then Node can run.

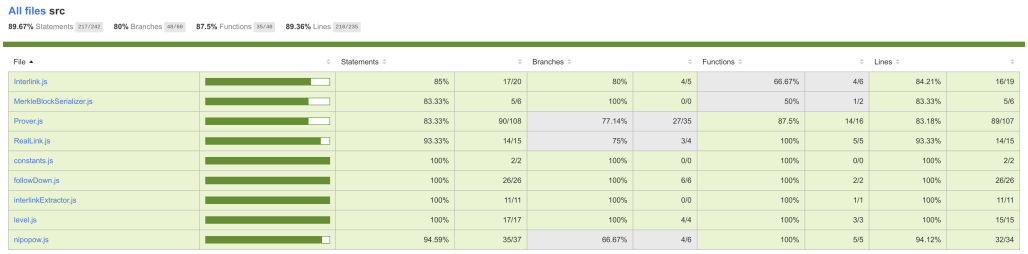

Figure 4.1: Snapshot from the test coverage report for the prover codebase.

Another is that all the external libraries we use, including bcash are not written including Flow types. Flow does provide a repository with type definitions for popular packages, however due to the niche category of our dependencies only a few were covered. This meant that the burden was on us to write type definitions for our dependencies. We had to manually write type definitions for bcash and buffer-map. Thankfully, there is previous work [1] on type definitions for bcoin which we were able to utilize and extend since the APIs are really similar as bcash is a fork of bcoin.

4.15 Testing

Due to the real-time and asynchronous nature of the prover it's important that our implementation is resilient and bug-free. In order to assert that the code has been thoroughly unit tested, boasting a code coverage of 90% as shown in Figure 4.1. The tests are written and run with Jest [16]. There are also integration tests using real-world data.

[1] Available at https://github.com/OrfeasLitos/TrustIsRisk.js/blob/247c8b182f5bb4bed11df0d3b9136a3e27848798/flow-typed/npm/bcoin_vx.x.x.js

Chapter 5

Conclusion

5.1 Summary

In this work we have studied the state-of-the-art constructions for creating proofs in the Prover-Verifier Model. We adapted the most fit construction for the purpose, NIPoPoWs, for the Bitcoin setting, by utilizing the User Activated Velvet Fork deployment strategy outlined in the NIPoPoWs paper and sidestepping the constant difficulty requirement.

We designed and implemented the specifics of the Velvet Fork, including methods for encoding the interlink. We paid close attention to the consumption of the interlinks, making them easily consumable by lite nodes by utilizing SPV tagged outputs.

We provided two implementations for the **interlinker**, an implementation which requires a full-node and one which is based on a lite-node with SPV. These implementations actually create the velvet fork and can be easily adapted to fork on any Bitcoin-based cryptocurrency, as well as perform any other kind of velvet fork. They are both released as open-source and are production-grade software.

We deployed the interlinker implementations on the Bitcoin Cash testnet, closely monitoring their performance and reliability, while fixing any anomalies due to bugs in our software as they developed.

We provided cost estimates for deploying our User-Activated Velvet Fork on Bitcoin Cash mainnet, where it can be seen that even though one needs to spend one transaction per block, due to the low fees on Bitcoin Cash is incredibly manageable even for individuals.

We analyzed blockchain data from Bitcoin and Bitcoin Cash, on their mainnets and testnets and discovered that the interlink encoding could be improved. We provided an optimization for interlink encoding based on our observations called **interlink compression**. We provided statistics which indicate how much of an improvement such a change can bring on real-world blocks.

We provided an implementation for the **prover**, which is based on a lite node. The prover is able to interact in the Bitcoin network, and has knowledge of the velvet fork, as well as of the actual interlink of each block. It is capable of discovering the velvet transactions and recovering the interlink commitments from them, and knowing if a block has a valid interlink or not. It is capable of creating both suffix and infix proofs for Bitcoin Cash.

5.2 Proof Consumption

The velvet fork and prover together enable any user to create NIPoPoW proofs on Bitcoin Cash. These proofs can be utilized as part of many applications. We will now look at some potential applications of very high interest to the cryptocurrency community.

5.2.1 Super-light Clients

Currently lite clients operate by accepting multiple SPV proofs. As we have seen these proofs are linear in chain size. One can very easily imagine modified light clients which have the ability to

verify and compare NIPoPoW proofs instead of SPV proofs in order to discover the longest chain or whether a transaction is included in the longest chain.

5.2.2 Cross-chain Transactions

Another popular application is cross-chain transactions: for example sending an amount of Bitcoin to Ethereum trustlessly. Currently when one wishes to move from one cryptocurrency to the other they usually utilize a centralized exchange. This is both time-consuming and trustless.

There is also the issue of the denomination: one would usually exchange their Bitcoin for an equivalent amount of Ethereum, according to the market rate. While in some cases this is desired, it is usually the case that one desires to transfer their funds to another cryptocurrency but keep them in the original denomination.

To solve this issue there are tokens in the Ethereum network like WBTC [25] which claim to offer an ERC20 token pegged 1:1 to Bitcoin, but have the need for a trusted third party called a custodian who is entrusted to generate tokens at will. It should be obvious that such solutions are not trustless.

An alternative would be to offer such a token but have the contract issue an amount of tokens if and only if it can can be sufficiently proven that the actor who requested the issuance has burned the same amount of Bitcoin on the other chain. Many constructions of this sort have been explored in the literature [34], including most importantly a construction using NIPoPoWs by Kiayias and Zindros [19]. An implementation of this construction in Solidity for the Ethereum chain has been provided by Christoglou [7].

5.3 Future Work

5.3.1 Bitcoin Cash mainnet Deployment

In order to make our work really useful to the general public the obvious step would be to deploy our interlinker on the mainnet network of Bitcoin Cash. This is a straightforward process, and it is directly supported by the software as is. We note that due to the low fees of Bitcoin Cash and consequently the low cost for operating an interlinker, it is very accessible for anyone to operate such a fork.

5.3.2 Interlink Compression Implementation

While we propose interlink compression, the development was very recent and has not been yet integrated in our interlinkers and prover. This change can be implemented and deployed in a backwards-compatible manner, meaning that the existing interlinks will not be rendered invalid. The changes to the interlinker implementation are fairly straightforward: only the interlink encoding would have to change. However, the prover has to support both the block list and block set constructions, and prefer the block set interlinks where available.

5.3.3 Bitcoin Deployment

The usefulness of NIPoPoWs is not limited to Bitcoin Cash only. The software can be easily deployed on any Bitcoin-based cryptocurrency, including Bitcoin itself, only requiring minimal changes.

5.3.4 Implementation For Other Major Cryptocurrencies

Other than Bitcoin-based cryptocurrencies, NIPoPoWs work for any Proof-of-Work cryptocurrency. We think velvet forks like the one outlined in this work would be viable and incredibly beneficial for many other cryptocurrencies, most notably Ethereum, Ethereum Classic, and Dash.

5.3.5 Verifier Smart Contract Implementation

The Proof-of-Work sidechains implementation by Christoglou [7] outlined earlier is unable to verify proofs created from velvet forked chains like the ones our provers create. It is therefore necessary for the implementation to be modified in order to support such proofs. Such a contract, if deployed on the Ethereum chain would allow users to trustlessly transfer Bitcoin Cash to the Ethereum chain by making use of proofs our prover can generate for the Bitcoin Cash chain.

Bibliography

[1] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. O'Reilly Media, Inc., 2014.

[2] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[3] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[4] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *URL: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 10/09/2016)*, 2014.

[6] Alexander Chepurnoy. Ergo platform, 2017. URL: https://ergoplatform.org/.

[7] Georgios Christoglou. Enabling crosschain transactions using nipopows. Master's thesis, Imperial College London, 2018.

[8] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[9] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[10] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[11] Jeff Garzik. jgarzik/python-bitcoinrpc: Python interface to bitcoin's json-rpc api, 2013. URL: https://github.com/jgarzik/python-bitcoinrpc.

[12] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[13] Harald Hoyer. Allow op_return with two pushdata by haraldh · pull request #6396 · bitcoin/bitcoin, 2015. URL: https://github.com/bitcoin/bitcoin/pull/6396.

[14] Docker Inc. docker/compose: Define and run multi-container applications with docker, 2013. URL: https://github.com/docker/compose/.

[15] Facebook Inc. Flow: A static type checker for javascript, 2014. URL: https://flow.org/.

[16] Facebook Inc. Jest: Delightful javascript testing, 2014. URL: https://jestjs.io.

[17] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 61–78. Springer, 2016.

[18] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. Technical report, Cryptology ePrint Archive, Report 2017/963, 2017. Accessed: 2017-10-03, 2017.

[19] Aggelos Kiayias and Dionysis Zindros. Proof-of-work sidechains. 2018.

[20] Holger Krekel. pytest: simple powerful testing with python, 2004. URL: https://pypi.org/project/pytest/.

[21] Yehuda Lindell and Jonathan Katz. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[22] Roman Matzutt, Jens Hiller, Martin Henze, Jan Henrik Ziegeldorf, Dirk Müllmann, Oliver Hohlfeld, and Klaus Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018.

[23] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[24] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[25] Andrew Miller. Wbtc wrapped bitcoin an erc20 token backed 1:1 with bitcoin, 2019. URL: https://www.wbtc.network/.

[26] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[27] Kenneth Reitz. Python development workflow for humans, 2017. URL: https://pypi.org/project/pipenv/.

[28] Bitcoin Team. Developer guide - bitcoin, 2017. URL: https://bitcoin.org/en/developer-guide.

[29] Nimiq Team. Nimiq, 2018. URL: https://nimiq.com/en/.

[30] WebDollar Team. Webdollar - currency of the internet, 2017. URL: https://webdollar.io.

[31] Peter Todd. petertodd/python-bitcoinlib: Python2/3 library providing an easy interface to the bitcoin data structures and protocol, 2014. URL: https://github.com/petertodd/python-bitcoinlib.

[32] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[33] A Zamyatin, N Stifter, A Judmayer, P Schindler, E Weippl, and WJ Knottebelt. A wild velvet fork appears! inclusive blockchain protocol changes in practice. In *5th Workshop on Bitcoin and Blockchain Research, Financial Cryptography and Data Security*, volume 18, 2018.

[34] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William J Knottenbelt. Xclaim: Interoperability with cryptocurrency-backed tokens. Technical report, Cryptology ePrint Archive, Report 2018/643, 2018. https://eprint. iacr. org ....