National and Kapodistrian University of Athens
School of Science

Department of Informatics and Telecommunications

Postgraduate Studies
Computer Systems: Software and Hardware

Master Thesis

A Gas-efficient Superlight Bitcoin Client in Solidity

Stelios Daveas

Supervisors: **Aggelos Kiayias**, Associate Professor, University of Edinburgh
Dionysis Zindros, PhD, University of Athens
Kostis Karantias, BSc, University of Ioannina

Athens,
June 2020

Master Thesis

A Gas-efficient Superlight Bitcoin Client in Solidity

Stelios Daveas
Reg.Nr.: M1597

Supervisors:

Aggelos Kiayias, Associate Professor, University of Edinburgh
Dionysis Zindros, PhD, University of Athens
Kostis Karantias, BSc, University of Ioannina

Thesis Committee:

Aggelos Kiayias, Associate Professor, University of Edinburgh
Mema Roussopoulos, Associate Professor, University of Athens
Yannis Smaragdakis, Professor, University of Athens

Abstract

During the last years, significant effort has been put into enabling blockchain interoperability, and several protocols have been proposed towards establishing crosschain communication. Most notably, superlight clients, a new generation of verifiers, have emerged. These clients demand only a poly-logarithmic number of block headers in order to verify transactions, rather than the entire span of the underlying chain. Albeit these constructions have been established theoretically, no practical implementation exists to date. In this paper, we focus on Non-Interactive Proofs of Proof of Work (NIPoPoWs), a probabilistic structure based on superblocks that is provably secure and provides succinct proofs of events in a blockchain. In particular, we discuss a gas-efficient implementation for the verification of NIPoPoWs in Solidity, enabling crosschain events from Bitcoin to Ethereum. We explore patterns and techniques that considerably reduce gas consumption, and may have applications to other smart contracts. We introduce a pattern that we term "hash-and-resubmit" that eliminates persistent storage almost entirely, leading to significant increase of performance. Furthermore, we alleviate the burden of expensive on-chain operations, which we transfer off-chain, and we make use of an optimistic schema that replaces functionalities of linear complexity with constant operations. Lastly, we make a cryptoeconomic analysis, and set concrete values regarding the cost that comes with using our client. Our implementation in Solidity is accompanied by thorough unit tests. We display the performance gain of our solution compared to previous work, and we mitigate security issues we encountered such as premining.

Contents

Chapter 1

Introduction

Bitcoin [33] is a form of decentralized money. Before Bitcoin was invented, the only way to use money digitally was through an intermediary like a bank. However, Bitcoin changed this by creating a decentralized form of currency that individuals can trade directly without the need for an intermediary. Each Bitcoin transaction is validated and confirmed by the entire Bitcoin network. There is no single point of failure so the system is virtually impossible to shut down, manipulate or control.

The person (or group of people, as many think) behind Bitcoin, is known by the name Shatoshi Nakamoto. Shatoshi put forth a construction that nowadays some consider one of the most important achievements of our age, all fitting into a 9-page paper. Bitcoin was published in November 2008, shortly followed by the initiation of the Bitcoin network in January 2009 and is the first ever secure and trust-less currency.

One of the by-products of the Bitcoin is blockchain. Blockchain technology was created by fusing already existing technologies like cryptography, proof of work and decentralized network architecture together in order to create a system that can reach decisions without a central authority. There was no "blockchain technology" before Bitcoin was invented, but once Bitcoin became a reality, people started noticing how and why it works and named this construction blockchain. Blockchain constitutes the very core of Bitcoin. Later, it was realized that a currency like Bitcoin is just one of the utilizations of the blockchain technology.

Ethereum [42, 5] was first proposed in late 2013 and then brought to life in 2014. Ethereum is a blockchain network that, apart from its digital currency, Ether, hosts decentralized programs. These decentralized apps (Dapps), or smart contracts, are written in Solidity [1], the programming language of Ethereum and yield to no single person control, not even to their author. The Ethereum platform is fully decentralized and consists of thousands of independent computers running it. Once a program is deployed to the Ethereum network it will be executed as written, hence the famous phrase: "code is law". Ethereum is a network of computers that together combine into one powerful, decentralized supercomputer. Ethereum is often characterized as the second era of blockchain networks.

1.1 Motivation

With the passing of time, new cryptocurrencies, altcoins as they are called in the cryptocurrency folklore, are created every day. Some altcoins bring new features to the cryprocurrency market and are accepted by the community, even becoming popular. As of April 2020, there were over 5.392 cryptocurrencies with a total market capitalisation of $201 billion.

A newcomer to this world of distributed coins would possibly expect that there must be some kind of established protocol for all these distinct blockchain to interact; a way for Alice, who keeps her funds in Bitcoins, to transfer an amount to Bob, who keeps his funds in Ether

and vice-versa[1]. In reality, the problem of blockchain interoperability had not been researched until recently, and, to date, there is still no commonly accepted decentralized protocol that enables interactions across blockchains, the so-called crosschain operations.

In general, crosschain-enabled blockchains A, B would satisfy the following:

- Crosschain trading: Alice with deposits in blockchain A, can make a payment to Bob at blockchain B.
- Crosschain fund transfer: Alice can transfers her funds from blockchain A to blockchain B. After this operation, the funds no longer exist in blockchain A. Alice can later decide to transfer any portion of the original amount to the blockchain of origin.

Currently, crosschain operations are only available to the users via third-party applications, such as multi-currency wallets. It is obvious that this centralized treatment opposes the nature of the blockchain and the introspective of decentralized currencies. This contradiction motivated us to create a solution that enables cheap and trust-less crosschain operations.

1.2 Rationale

In order to perform crosschain operations, mechanism that allows users of blockchain A to discover events that have occurred in chain B, such as settled transactions, must be introduced. One tricky aspect is to ensure the atomicity of such operations, which require that either the transactions take place in *both* chains, or in *neither*. This is achievable through atomic swaps [34, 24]. However, atomic swaps provide limited functionality in that they do not allow the generic transfer of information from one blockchain to a smart contract in another. For many applications, a richer set of functionalities is needed [28, 25]. To communicate the fact that an event took place in a source blockchain, a naïve approach is to have users relay all the source blockchain blocks to a smart contract residing in the target chain, which functions as a client for the remote chain and validates all incoming information [11]. This approach, however, is impractical because a sizable amount of storage is needed to host entire chains as they grow in time. As of June 2020, Bitcoin [33] chain spans roughly 245 GB, and Ethereum [42, 5] has exceeded 300 GB[2].

One early solution to compress the extensive size of blockchain and improve the efficient of a client is addressed by Nakamoto [33] with the Simplified Payment Verification (SPV) protocol. In SPV, only the headers of blocks are stored, saving a considerable amount of storage. However, even with this protocol, the process of downloading and validating all block headers still demands a considerable amount of resources since they grow linearly in the size of the blockchain. In Ethereum, for instance, headers sum up to approximately 4.8 GB[3] of data. These numbers quickly become impractical when it comes to consuming and storing the data within a smart contract.

Another idea to make chain A interact with chain B, is to provide a cryptographic proof to chain B that an event occurred in chain A. Secure cryptographic proofs are mathematical constructions that are easy to verify and impossible for an adversary to forge and are broadly used in cryptography and blockchain in particular. In order to be more efficient than SPV, the size of these proof needs to be small related to the size of the blockchain. This way, we are be able to create proofs for events in chain A and send it to chain B for validation. If chain B supports smart contracts, like Ethereum, the proof can be verified automatically and transparently *on-chain*. Notice that no third-party is involved through the entire process.

[1] The transfer of an amount from one chain to another is called one-way peg, and the transfer of funds back to the original chain is called two-way peg.

[2] Size of blockchain derived from https://www.statista.com, https://etherscan.io

[3] Calculated as the number of blocks (10,050,219) times the size of header (508 bytes). Statistics by https://etherscan.io/.

1.3 Related Work

NIPoPoWs were introduced by Kiayias, Miller and Zindros [27] and their application to cross-chain communication was described in follow-up work [28], but only theoretically and without providing an implementation. A few cryptocurrencies already include built-in NIPoPoWs support, namely ERGO [9], Nimiq [38], and WebDollar [39]; these chains can natively function as *sources* in cross-chain protocols.

Christoglou [12] provided a Solidity smart contract which is the first implementation of crosschain events verification based on NIPoPoWs, where proofs are submitted and checked for their validity, marking the first implementation of an *on-chain* verifier. This solution, however, is impractical due to extensive usage of resources, widely exceeding the Ethereum block gas limit.

Other attempts have been made to address the verification of Bitcoin transactions to the Ethereum blockchain, most notably BTC Relay [11], which requires storing a full copy of all Bitcoin block headers within the Ethereum chain.

1.4 Our contributions

Notably, no practical implementation for an on-chain superlight clients exists to date. In this paper, we focus on constructing a practical client for superblock NIPoPoWs. For the implementation of our client, we refine the NIPoPoW protocol based on a series of keen observations. These refinements allow us to leverage useful techniques that construct a practical solution for proof verification. We believe this achievement is a decisive and required step towards establishing NIPoPoWs as the standard protocol for cross-chain communication. A summary of our contributions in this paper is as follows:

1. We develop the first on-chain decentralized client that securely verifies crosschain events and is practical. Our client establishes a trustless and efficient solution to the interoperability problem. We implement[4] our client in Solidity, and we verify Bitcoin events to the Ethereum blockchain. The security assumptions we make are no other than SPV's [23, 43].

2. We present a novel pattern which we term *hash-and-resubmit*. Our pattern significantly improves performance of Ethereum smart contracts [42, 5] in terms of gas consumption by utilizing the *calldata* space of Ethereum blockchain to eliminate high-cost storage operations.

3. We create an *optimistic* schema which we incorporate into the design of our client. This design achieves significant performance improvement by replacing linear complexity verification of proofs with constant complexity verification.

4. We demonstrate that superblock NIPoPoWs are practical, making it the first efficient cross-chain primitive.

5. We present a cryptoeconometric analysis of NIPoPoWs. We provide concrete values for the collateral/liveness trade-off.

Our implementation meets the following requirements:

1. **Security**: The client implements a provably secure protocol.

2. **Decentralization**: The client is not dependent on trusted third-parties and operates in a transparent, decentralized manner.

3. **Efficiency**: The client complies with environmental constraints, i.e. block gas limit and calldata size limit of the Ethereum blockchain.

[4] Our implementation, unit tests and experiments can be found in `https://github.com/sdaveas/nipopow-verifier`.

We selected Bitcoin as the source blockchain as it the most popular cryptocurrency, and enabling crosschain transactions in Bitcoin is beneficial to the majority of the blockchain community. We selected Ethereum as the target blockchain because, besides its popularity, it supports smart contracts, which is a requirement in order to perform on-chain verification. We note here that prior to Bitcoin events being consumable in Ethereum, Bitcoin requires a velvet fork [44], a matter treated in a separate line of work [35].

Some applications that demonstrate the usage of our client are:

- Application #1
- Application #2
- Application #3

1.5 Structure

Refine this

In Section 2 we describe the blockchain technologies that are relevant to our work. In Section 3 we put forth the *hash-and-resubmit* pattern. We demonstrate the improved performance of smart contracts using the pattern, and how it is incorporated into our client. In Section 4, we present an alteration to the NIPoPoW protocol that enables the elimination of look-up structures. This allows for efficient interactions due to the considerably smaller size of dispatched proofs. In Section 5, we put forth an optimistic schema that significantly lowers the complexity of a proof's structural verification from linear to constant, by introducing a new interaction which we term *dispute phase*. Furthermore, we present a technique that leverages the dispatch of a constant number of blocks in the contest phase. Finally, in Section 6, we present our cryptoeconomic analysis on our client and establish the monetary value of collateral parameters.

Chapter 2

Background

2.1 Cryptographic Hash Functions

A cryptographic hash function is a hash function that is suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit string of a fixed size (the "hash value", "hash", or "message digest") and is a one-way function, that is, a function which is practically infeasible to invert. Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Cryptographic hash functions are a basic tool of modern cryptography.

The ideal cryptographic hash function has the following main properties:

1. it is deterministic, meaning that the same message always results in the same hash
2. it is quick to compute the hash value for any given message
3. it is infeasible to generate a message that yields a given hash value
4. it is infeasible to find two different messages with the same hash value
5. a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value (avalanche effect)

2.2 Merkle Trees

A Merkle Tree [32], also known as a Binary Hash Tree is a data structure used for efficiently summarizing and verifying the integrity of large sets of data. Merkle Trees are binary trees containing cryptographic hashes. The term "tree" is used in computer science to describe a branching data structure, but these trees are usually displayed upside down with the "root" at the top and the "leaves" at the bottom of a diagram, as you will see in the examples that follow.

Merkle trees are used in Bitcoin to summarize all the transactions in a block, producing an overall digital fingerprint of the entire set of transactions, providing a very efficient process to verify if a transaction is included in a block. A Merkle Tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the root, or merkle root. The cryptographic hash algorithm used in Bitcoin's Merkle Trees is SHA256 applied twice, also known as double-SHA256.

When N data elements are hashed and summarized in a Merkle Tree, you can check to see if any one data element is included in the tree with at most 2*log2(N) calculations,making this a very efficient data structure.

The Merkle Tree is constructed bottom-up. In the example below, we start with four transactions A, B, C and D, which form the leaves of the Merkle Tree, shown in the diagram

at the bottom. The transactions are not stored in the Merkle Tree, rather their data is hashed and the resulting hash is stored in each leaf node as H_A, H_B, H_C and H_D:

$$\mathsf{H}_A = \mathsf{SHA256}(\mathsf{SHA256}(\text{Transaction A}))$$

Consecutive pairs of leaf nodes are then summarized in a parent node, by concatenating the two hashes and hashing them together. For example, to construct the parent node H_{AB}, the two 32-byte hashes of the children are concatenated to create a 64-byte string.That string is then double-hashed to produce the parent node's hash:

$$\mathsf{H}_{AB} = \mathsf{SHA256}(\mathsf{SHA256}(\mathsf{H}_A + \mathsf{H}_B))$$

The process continues until there is only one node at the top, the node known as the Merkle Root. That 32-byte hash is stored in the block header and summarizes all the data in all four transactions. This is displayed in Figure 2.1

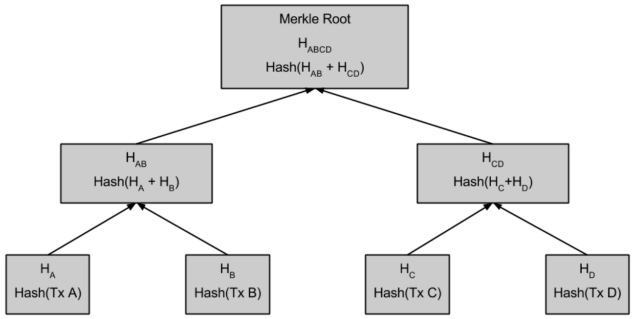


Figure 2.1: Merkle Tree

2.3 Blockchain

A blockchain is a decentralized, distributed, and oftentimes public, digital ledger consisting of records called blocks that is used to record transactions across many computers so that any involved block cannot be altered retroactively, without the alteration of all subsequent blocks. This allows the participants to verify and audit transactions independently and relatively inexpensively. A blockchain database is managed autonomously using a peer-to-peer network and a distributed timestamping server. They are authenticated by mass collaboration powered by collective self-interests. Such a design facilitates robust workflow where participants' uncertainty regarding data security is marginal. The use of a blockchain removes the characteristic of infinite reproducibility from a digital asset. It confirms that each unit of value was transferred only once, solving the long-standing problem of double spending. A blockchain has been described as a value-exchange protocol. A blockchain can maintain title rights because, when properly set up to detail the exchange agreement, it provides a record that compels offer and acceptance.

2.3.1 Blocks

Blocks hold batches of valid transactions that are hashed and encoded into a Merkle tree. Each block includes the cryptographic hash of the prior block in the blockchain, linking the two. The linked blocks form a chain. This iterative process confirms the integrity of the previous block, all the way back to the original genesis block. Sometimes separate blocks can be produced concurrently, creating a temporary fork. In addition to a secure hash-based history, any blockchain has a specified algorithm for scoring different versions of the history so that one with a higher score can be selected over others. Blocks not selected for inclusion

in the chain are called orphan blocks. Peers supporting the database have different versions of the history from time to time. They keep only the highest-scoring version of the database known to them. Whenever a peer receives a higher-scoring version (usually the old version with a single new block added) they extend or overwrite their own database and retransmit the improvement to their peers. There is never an absolute guarantee that any particular entry will remain in the best version of the history forever. Blockchains are typically built to add the score of new blocks onto old blocks and are given incentives to extend with new blocks rather than overwrite old blocks. Therefore, the probability of an entry becoming superseded decreases exponentially as more blocks are built on top of it, eventually becoming very low. For example, Bitcoin uses a proof-of-work system, where the chain with the most cumulative proof-of-work is considered the valid one by the network. There are a number of methods that can be used to demonstrate a sufficient level of computation. Within a blockchain the computation is carried out redundantly rather than in the traditional segregated and parallel manner.

2.3.2 Block Time

The block time is the average time it takes for the network to generate one extra block in the blockchain. Some blockchains create a new block as frequently as every five seconds. By the time of block completion, the included data becomes verifiable. In cryptocurrency, this is practically when the transaction takes place, so a shorter block time means faster transactions. The block time for Ethereum is set to between 14 and 15 seconds, while for Bitcoin it is on average 10 minutes.

2.4 Bitcoin

Bitcoin [33] is a cryptocurrency. It is a decentralized digital currency without a central bank or single administrator that can be sent from user to user on the peer-to-peer Bitcoin network without the need for intermediaries.

Transactions are verified by network nodes through cryptography and recorded in a public distributed ledger called a blockchain. Bitcoin was invented in 2008 by an unknown person or group of people using the name Satoshi Nakamoto and started in 2009 when its source code was released as open-source software. Bitcoins are created as a reward for a process known as mining. They can be exchanged for other currencies, products, and services.

The Bitcoin blockchain is displayed in Figure 2.2.

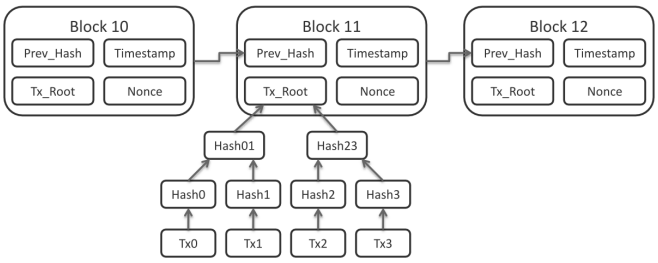


Figure 2.2: The Bitcoin blockchain

2.4.1 Transactions

Transactions are defined using a Forth-like[1] scripting language. Transactions consist of one or more inputs and one or more outputs. When a user sends Bitcoins, the user designates each address and the amount of Bitcoin being sent to that address in an output. To prevent

[1] https://www.taygeta.com/forth/dpans.html

double spending, each input must refer to a previous unspent output in the blockchain. The use of multiple inputs corresponds to the use of multiple coins in a cash transaction. Since transactions can have multiple outputs, users can send Bitcoins to multiple recipients in one transaction. As in a cash transaction, the sum of inputs (coins used to pay) can exceed the intended sum of payments. In such a case, an additional output is used, returning the change back to the payer. Any input satoshis not accounted for in the transaction outputs become the transaction fee.

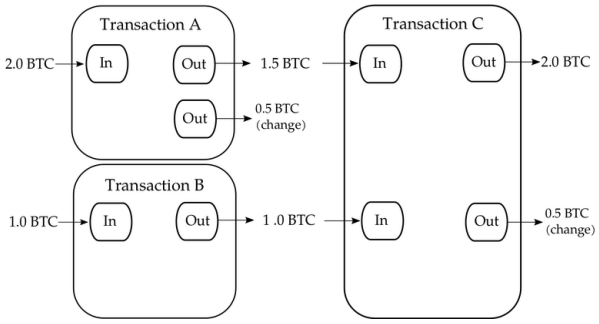


Figure 2.3: Bitcoin transactions

2.4.2 Mining

Early Bitcoin miners used GPUs for mining, as they were better suited to the proof-of-work algorithm than CPUs. Later amateurs mined Bitcoins with specialized FPGA and ASIC chips. The chips pictured have become obsolete due to increasing difficulty. Today, Bitcoin mining companies dedicate facilities to housing and operating large amounts of high-performance mining hardware. Semi-log plot of relative mining difficulty

Mining is a record-keeping service done through the use of computer processing power. Miners keep the blockchain consistent, complete, and unalterable by repeatedly grouping newly broadcast transactions into a block, which is then broadcast to the network and verified by recipient nodes. Each block contains a SHA-256 cryptographic hash of the previous block, thus linking it to the previous block and giving the blockchain its name.

To be accepted by the rest of the network, a new block must contain a proof-of-work (PoW). The system used is based on Adam Back's 1997 anti-spam scheme, Hashcash [2]. The PoW requires miners to find a number called a nonce, such that when the block content is hashed along with the nonce, the result is numerically smaller than the network's difficulty target. This proof is easy for any node in the network to verify, but extremely time-consuming to generate, as for a secure cryptographic hash, miners must try many different nonce values (usually the sequence of tested values is the ascending natural numbers: 0, 1, 2, 3, ... , 8) before meeting the difficulty target.

Every 2,016 blocks (approximately 14 days at roughly 10 min per block), the difficulty target is adjusted based on the network's recent performance, with the aim of keeping the average time between new blocks at ten minutes. In this way the system automatically adapts to the total amount of mining power on the network. Between 1 March 2014 and 1 March 2015, the average number of nonces miners had to try before creating a new block increased from 16.4 quintillion to 200.5 quintillion.

The proof-of-work system, alongside the chaining of blocks, makes modifications of the blockchain extremely hard, as an attacker must modify all subsequent blocks in order for the modifications of one block to be accepted. As new blocks are mined all the time, the difficulty of modifying a block increases as time passes and the number of subsequent blocks (also called confirmations of the given block) increases.

2.4.3 Simple payment verification - SPV

The Bitcoin blockchain has generated more than 600,000 blocks since its genesis[2] block. At the time of writing, a full node needs about 250GB of disk space to store the whole blockchain. Suppose a Bitcoin user wants to verify that their transaction has been mined successfully to the main chain. The only way to do it is to check the history of the transactions since the genesis block. Of course, the user only cares about their transactions and not every transaction that has ever occurred in the blockchain.

Satoshi's paper describes a method called the *Simple payment verification*(SPV) which tackles the problem in an efficient manner. Instead of downloading the whole blockchain, the user can request from a full node only the block headers. The size of all of the block headers of the main chain to date is around 80MB which is easily manageable for modern computers and smart phones. In order to avoid connecting to malicious nodes, it is often recommended for users to connect to multiple nodes and request information until they are convinced they have the longest chain. The longest chain is the chain with the most proof of work and not the chain with the most blocks. To prove that the network has accepted the transaction, the user has to link the transaction to a certain block in the main chain and ensure that blocks are added to the chain which means that more work is added to the chain.

2.5 Ethereum

Ethereum is an open source, public, blockchain-based distributed computing platform featuring smart contract (scripting) functionality. Ether is the cryptocurrency generated by the Ethereum platform as a reward to mining nodes for computations performed and is the only currency accepted in the payment of transaction fees on the platform. Ethereum is the second-largest cryptocurrency platform by market capitalization, behind Bitcoin.

Ethereum provides a decentralized virtual machine, the Ethereum Virtual Machine (EVM), which can execute scripts using an international network of public nodes. The virtual machine's instruction set, in contrast to others like Bitcoin Script, is Turing-complete. "Gas", an internal transaction pricing mechanism, is used to mitigate spam and allocate resources on the network.

Ethereum was proposed in late 2013 by Vitalik Buterin, a cryptocurrency researcher and programmer. Development was funded by an online crowdsale that took place between July and August 2014. The system then went live on 30 July 2015, with 72 million coins minted. This accounts for about 65 percent of the total circulating supply in April 2020.

2.5.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts in Ethereum. It is a 256-bit register stack, designed to run the same code exactly as intended. It is the fundamental consensus mechanism for Ethereum. The formal definition of the EVM is specified in the Ethereum Yellow Paper. Ethereum Virtual Machines have been implemented in C++, C#, Go, Haskell, Java, JavaScript, Python, Ruby, Rust, Elixir, Erlang.

2.5.2 Smart Contracts

Ethereum's smart contracts are based on different computer languages, which developers use to program their own functionalities. Smart contracts are high-level programming abstractions that are compiled down to EVM bytecode and deployed to the Ethereum blockchain for execution. They can be written in Solidity (a language library with similarities to C and JavaScript), Serpent (similar to Python, but deprecated), LLL (a low-level Lisp-like language), and Mutan (Go-based, but deprecated). There is also a research-oriented language under development called Vyper (a strongly-typed Python-derived decidable language).

[2]Genesis block is the first block mined in the blockchain

One issue related to using smart contracts on a public blockchain is that bugs, including security holes, are visible to all but cannot be fixed quickly. One example of this is the 17 June 2016 attack on The DAO, which could not be quickly stopped or reversed.

There is ongoing research on how to use formal verification to express and prove non-trivial properties. A Microsoft Research report noted that writing solid smart contracts can be extremely difficult in practice, using The DAO hack to illustrate this problem. The report discussed tools that Microsoft had developed for verifying contracts, and noted that a large-scale analysis of published contracts is likely to uncover widespread vulnerabilities. The report also stated that it is possible to verify the equivalence of a Solidity program and the EVM code.

2.5.3 EVM Implementations

Py-EVM is an evolving EVM which is created mainly for testing. The ease of access and use, the configuration freedom of its underlying test chain and its effectiveness for small size of data helped our first steps. However, as the input data size started to grow, the effectiveness of the tool rapidly fell.
Ganache is a popular EVM developed by the Truffle team. Its speed and configuration freedom are its main advantages. However, its extreme memory requirement made it impossible to use when the sizes of the input became analogous to the Bitcoin blockchain size.
Geth is another popular EVM which is created by the Ethereum team. It supports heavy customization while its memory usage is very limited compared to Ganache, even for extensive inputs. It has, however, higher execution times than Ganache for our purposes because Geth doesn't natively support auto-mining. That is the capability to mine new blocks only when new transactions are available. In order to avoid intense use of the CPU, we injected a function in Geth's `miner` object be only invoked when a new transaction is available. This, together with the fact that mining is probabilistic, put an extra overhead at the execution time.

2.6 Gas Profiling

One useful utility we used is solidity-gas-profiler[3], a profiling utility. This experimental software displays the gas usage in a smart contract for each line of code. It gave us great insights regarding the gas usage across contract's functions, and consequently helped us target the functionalities that needed to be refined. An example of the execution of the gas profiled is displayed in listing 2.1

```
1  // Gas used by transaction: 183443
2  0       pragma solidity ^0.6.6;
3  84      contract Contract {
4  0
5  0           uint s;
6  0           address holder;
7  0
8  0           constructor (address a) public {
9  0               holder = a;
10 0           }
11 0
12 89          function foo (uint c) public  {
13 598             for (uint i=0; i<c; i++ ) {
14 155069              s = i;
15 0               }
16 226             if (s > 1) {
17 5233                s += 1;
18 0               }
19 0           }
20 0       }
```

Listing 2.1: Gas profile

[3] https://github.com/yushih/solidity-gas-profiler

2.7 Non-Interactive Proofs Of Proof Of Work

2.7.1 Participants

We consider a setting where the blockchain network consists of two different types of nodes: The first kind, *full nodes*, are responsible for the maintenance of the chain including verifying it and mining new blocks. The second kind, *verifiers*, connect to full nodes and wish to learn facts about the blockchain without downloading it, for example whether a particular transaction is confirmed. The full nodes therefore also function as *provers* for the verifiers. Each verifier connects to multiple provers, at least one of which is assumed to be honest.

2.7.2 Blockchain Model

Each honest full node locally maintains a *chain* C, a sequence of blocks. In understanding that we are developing an improvement on top of SPV, we use the term *block* to mean what is typically referred to as a *block header*. Each block contains the Merkle Tree root [32] of transaction data $\overline{x}$, the hash s of the previous block in the chain known as the *previd*, as well as a nonce value *ctr*. As discussed in the Introduction, the compression of application data $\overline{x}$ is orthogonal to our goals in this paper and has been explored in independent work [10] which can be composed with ours. Each block $b = s \parallel \overline{x} \parallel ctr$ must satisfy the proof-of-work [14] equation $H(b) \leq T$ where T is a constant *target*, a small value signifying the difficulty of the proof-of-work problem. Our treatment is in the *static difficulty* case, so we assume that T is constant throughout the execution[4]. $H(B)$ is known as the *block id*.

Blockchains are finite block sequences obeying the *blockchain property*: that in every block in the chain there exists a pointer to its previous block. A chain is *anchored* if its first block is *genesis*, denoted $\mathcal{G}$, a special block known to all parties. This is the only block the verifier knows about when it boots up. For chain addressing we use Python brackets $\mathsf{C}[\cdot]$. A zero-based positive number in a bracket indicates the indexed block in the chain. A negative index indicates a block from the end, e.g., $\mathsf{C}[-1]$ is the tip of the blockchain. A range $\mathsf{C}[i{:}j]$ is a subarray starting from i (inclusive) to j (exclusive). Given chains $\mathsf{C}_1, \mathsf{C}_2$ and blocks A, Z we concatenate them as $\mathsf{C}_1\mathsf{C}_2$ or $\mathsf{C}_1 A$ (if clarity mandates it, we also use the symbol $\parallel$ for concatenation). Here, $\mathsf{C}_2[0]$ must point to $\mathsf{C}_1[-1]$ and A must point to $\mathsf{C}_1[-1]$. We denote $\mathsf{C}\{A{:}Z\}$ the subarray of the chain from block A (inclusive) to block Z (exclusive). We can omit blocks or indices from either side of the range to take the chain to the beginning or end respectively. As long as the blockchain property is maintained, we freely use the set operators $\cup$, $\cap$ and $\subseteq$ to denote operations between chains, implying that the appropriate blocks are selected and then placed in chronological order.

During every round, every party attempts to *mine* a new block on top of its currently adopted chain. Each party is given q queries to the random oracle which it uses in attempting to mine a new block. Therefore the adversary has tq queries per round while the honest parties have $(n-t)q$ queries per round. When an honest party discovers a new block, they extend their chain with it and broadcast the new chain. Upon receiving a new chain C' from the network, an honest party compares its length $|\mathsf{C}'|$ against its currently adopted chain C and adopts the newly received chain if it is longer. It is assumed that the honest parties control the majority of the computational power of the network. If so, the protocol ensures consensus among the honest parties: There is a constant k, the *Common Prefix* parameter, such that, at any round, all the chains belonging to honest parties share a common prefix of blocks; the chains can deviate only up to k blocks at the end of each chain [19]. Concretely, if at some round r two honest parties have C_1 and C_2 respectively, then either $\mathsf{C}_1[:-k]$ is a prefix of C_2 or $\mathsf{C}_2[:-k]$ is a prefix of C_1.

[4] A treatment of variable difficulty NIPoPoWs has been explored in the soft fork case [45], but we leave the treatment of velvet fork NIPoPoWs in the variable difficulty model for future work.

2.7.3 Superblocks

Some valid blocks satisfy the proof-of-work equation better than required. If a block b satisfies $H(b) \leq 2^{-\mu}T$ for some natural number $\mu \in \mathbb{N}$ we say that b is a *μ-superblock* or a block *of level μ*. The probability of a new valid block achieving level μ is $2^{-\mu}$. The number of levels in the chain will be $\log|\mathsf{C}|$ with high probability [26]. Given a chain C, we denote $\mathsf{C}{\uparrow}^{\mu}$ the subset of μ-superblocks of C.

Non-Interactive Proofs of Proof-of-Work (NIPoPoW) protocols allow verifiers to learn the most recent k blocks of the blockchain adopted by an honest full node without downloading the whole chain. The challenge lies in building a verifier who can find the suffix of the longest chain between claims of both honest and adversarial provers, while not downloading all block headers. Towards that goal, the *superblock* approach uses superblocks as samples of proof-of-work. The prover sends superblocks to the verifier to convince them that proof-of-work has taken place without actually presenting all this proof-of-work. The protocol is parametrized by a constant security parameter m. The parameter determines how many superblocks will be sent by the prover to the verifier and security is proven with overwhelming probability in m.

2.7.4 Prover

The prover is a friendly but untrusted node that submits proofs of events that happen on a *source* chain. These proofs are advertised to a different *target* chain and their validity is ensured by other honest nodes. Provers need to monitor both source and target chains in order to (a) create proofs of events from the source chain and (b) validate proofs that are submitted by other nodes to the target chain.

The prover selects various levels μ and, for each such level, sends a carefully chosen portion of its μ-level *superchain* $\mathsf{C}{\uparrow}^{\mu}$ to the verifier. In standard blockchain protocols such as Bitcoin and Ethereum, each block $\mathsf{C}[i+1]$ in C points to its previous block $\mathsf{C}[i]$, but each μ-superblock $\mathsf{C}{\uparrow}^{\mu}\ [i+1]$ does not point to its previous μ-superblock $\mathsf{C}{\uparrow}^{\mu}\ [i]$. It is imperative that an adversarial prover does not reorder the blocks within a superchain, but the verifier cannot verify this unless each μ-superblock points to its most recently preceding μ-superblock. The proposal is therefore to *interlink* the chain by having each μ-superblock include an extra pointer to its most recently preceding μ-superblock. To ensure integrity, this pointer must be included in the block header and verified by proof-of-work. However, the miner does not know which level a candidate block will attain prior to mining it. For this purpose, each block is proposed to include a pointer to the most recently preceding μ-superblock, for every μ. as illustrated in Figure 2.4. As these levels are only $\log|\mathsf{C}|$, this only adds $\log|\mathsf{C}|$ extra pointers to each block header.

Albeit a proof can be structurally correct, an adversary prover can dispatch a proof of an dishonest chain. This is addressed by establishing a contest period, in which honest provers can submit honest proofs that invalidate the originally submitted. The rational behind this is that the honest chain encapsulates the most proof-of-work, and thus achieves better score among other proofs that represent shorter, dishonest chains. To provide incentives for honest nodes to contest fraud proofs, a collateral is included to the submission of proofs which is paid to the party that achieves to present a better proofs than the originally submitted. If no contest happen, the collateral is returned to the original issuer.

2.7.5 Verifier

The verifier is an application that performs operations on proofs and does not need to be aware of the state of each chain. Upon receiving two proofs, π_1 and π_2, the NIPoPoW verifier first checks that they form valid chains. To check that they are valid chains, the verifier ensures every block in the proof contains a pointer to its previous block inside the proof through either the *previd* pointer in the block header, or in the interlink vector. If any of these checks fail, the proof is rejected. It then compares π_1 against π_2 using the $\leq_m$ operator, which works as follows. It finds the lowest common ancestor block $b = (\pi_1 \cap \pi_2)[-1]$; that is, b is the most

Algorithm 1 The Prove algorithm for the NIPoPoW protocol in a soft fork

function $\mathsf{Prove}_{m,k}(\mathsf{C})$
  $B \leftarrow \mathsf{C}[0]$ ▷ Genesis
  for $\mu = |\mathsf{C}[-k-1].\mathsf{interlink}|$ down to 0 **do**
    $\alpha \leftarrow \mathsf{C}[:-k]\{B:\}\uparrow^{\mu}$
    $\pi \leftarrow \pi \cup \alpha$
    if $m < |\alpha|$ **then**
      $B \leftarrow \alpha[-m]$
    end if
  end for
  $\chi \leftarrow \mathsf{C}[-k:]$
  return $\pi\chi$
end function

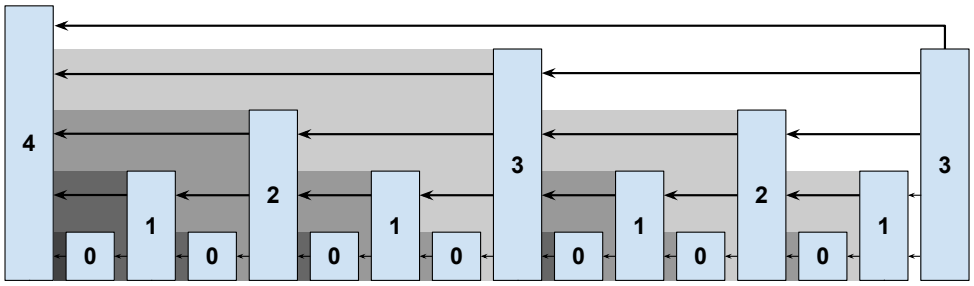


Figure 2.4: The interlinked blockchain. Each superblock is drawn taller according to its achieved level. Each block links to all the blocks that are not being overshadowed by their descendants. The most recent (right-most) block links to the four blocks it has direct line-of-sight to.

recent block shared among the two proofs. Subsequently, it chooses the level μ_1 for π_1 such that $|\pi_1\{b:\}\uparrow^{\mu_1}| \geq m$ (i.e., π_1 has at least m superblocks of level μ_1 following block b) and the value $2^{\mu_1}|\pi_1\{b:\}\uparrow^{\mu_1}|$ is maximized. It chooses a level μ_2 for π_2 in the same fashion. The two proofs are compared by checking whether $2^{\mu_1}|\pi_1\{b:\}\uparrow^{\mu_1}| \geq 2^{\mu_2}|\pi_2\{b:\}\uparrow^{\mu_2}|$ and the proof with the largest score is deemed the winner. The comparison is illustrated in Algorithm 2.

An adversary prover could skip the blocks of interest and present an honest and longer chain that is considered a better proof. For that reason, the last step of the algorithm in the suffix verifier is changed to not only store the best proof but also combine the two proofs by including all of the ancestor blocks of the losing proof. This is called infix verification and is guaranteed to include the blocks of interest. The resulting best proof is stored as a DAG(Directed Acyclic Graph), as in Algorithm 3.

The exact NIPoPoW protocol works like this: The prover holds a full chain C. When the verifier requests a proof, the prover sends the last k blocks of their chain, the suffix $\chi = \mathsf{C}[-k:]$, in full. From the larger prefix $\mathsf{C}[:-k]$, the prover constructs a proof π by selecting certain superblocks as representative samples of the proof-of-work that took place. The blocks are picked as follows. The prover selects the *highest* level μ^* that has at least m blocks in it and includes all these blocks in their proof (if no such level exists, the chain is small and can be sent in full). The prover then iterates from level $\mu = \mu^* - 1$ down to 0. For every level μ, it includes sufficient μ-superblocks to cover the last m blocks of level $\mu + 1$, as illustrated in Algorithm 1. Because the density of blocks doubles as levels are descended, the proof will contain in expectation $2m$ blocks for each level below μ^*. As such, the total proof size $\pi\chi$ will be $\Theta(m \log |\mathsf{C}| + k)$. Such proofs that are polylogarithmic in the chain size constitute an exponential improvement over traditional SPV clients and are called *succinct*.

Algorithm 2 The Verify algorithm for the NIPoPoW protocol

function best-arg$_m(\pi, b)$
  $M \leftarrow \{\mu : |\pi\uparrow^\mu \{b :\}| \geq m\} \cup \{0\}$ ▷ Valid levels
  return $\max_{\mu \in M}\{2^\mu \cdot |\pi\uparrow^\mu \{b :\}|\}$ ▷ Score for level
end function
operator $\pi_A \geq_m \pi_B$
  $b \leftarrow (\pi_A \cap \pi_B)[-1]$ ▷ LCA
  return best-arg$_m(\pi_A, b) \geq$ best-arg$_m(\pi_B, b)$
end operator
function Verify$^Q_{m,k}(\mathcal{P})$
  $\tilde{\pi} \leftarrow$ (Gen) ▷ Trivial anchored blockchain
  for $(\pi, \chi) \in \mathcal{P}$ **do** ▷ Examine each proof in $\mathcal{P}$
    if 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

Algorithm 3 The verify algorithm for the NIPoPoW infix protocol

function ancestors$(B,$ blockById$)$
  if $B =$ Gen **then**
    return $\{B\}$
  end if
  $\mathsf{C} \leftarrow \emptyset$
  for id $\in B$.interlink **do**
    if id $\in$ blockById **then**
      $B' \leftarrow$ blockById[id] ▷ Collect into DAG
      $\mathsf{C} \leftarrow \mathsf{C} \cup$ ancestors$(B',$ blockById$)$
    end if
  end for
  return $\mathsf{C} \cup \{B\}$
end function
function verify-infx$^D_{\ell,m,k}(\mathcal{P})$
  blockById $\leftarrow \emptyset$
  for $(\pi, \chi) \in \mathcal{P}$ **do**
    for $B \in \pi$ **do**
      blockById[id(B)] $\leftarrow B$
    end for
  end for
  $\tilde{\pi} \leftarrow$ best $\pi \in \mathcal{P}$ according to suffix verifier
  return $D($ancestors$(\tilde{\pi}[-1],$ blockById$))$
end function

2.8 NIPoPoW Verifier in Solidity

2.8.1 Methodology

Christogloy et. al. has provided a Solidity implementation. We used this implementation as a basis for our work. Since we adopted common primitives, we used some of the tools the authors used for functionalities such as constructing blockchains and proofs. For the purposes of our implementation, we to enhance the functionality of the existing tools in some cases. We are thankful to the writers for sharing their implementation. This greatly facilitated our work.

In this subsection, we describe the model of Non-Interactive Proofs of Proof of Work in the context of the verifier implementation in Solidity. This includes the following:

1. Construction of a blockchain.
2. Construction of a proof for an event in the blockchain.
3. Verification of the proof.

Blockchain

The tool that creates the blockchain has been created by Andrew Miller, one of the writers of Non-Interactive Proofs of Proof of Work paper. The tool is using the Bitcoin library[5] to construct a blockchain similar to Bitcoin's. The interlink pointers are organized into a Merkle Tree and the index is determined by their level. The Merkle root of the interlink tree is a 32-bit value, and is included in the block header as an additional value. The new size of the block header is 112 bytes. In order to ensure security, it is important for the interlink root to be included in the block header, as it is part of the proof. Otherwise, attackers could attack the proofs by reordering or including stray blocks. Miners can easily verify that the Merkle root is correct.

Superblock Levels

We assume that the difficulty target of mined blocks is constant. This is not the actual setting of the Bitcoin blockchain. The definition of superblocks is changed to a simpler definition and the level is determined by the number of leading zeros of the block header hash. Although this change does not take into account the difficulty target, the scoring of proofs does not generate security holes in the protocol.

Proof

The tool for the creation of proofs has also been created by Andrew Miller. The prover receives the following inputs:

- A blockchain with interlinks
- The security parameter k
- The security parameter m

2.8.2 Phases

The goal of the verifier is to securely determine if an event has occurred in the honest blockchain by examining proofs. A proof is submitted in combination with a predicate. The proof is considered valid if it is constructively correct and the predicate is evaluated against the submitted proof. The predicate represents the existence of an event in the source blockchain, such as the occurrence of a transaction. In this context, the predicate indicates the existence of a block within the proof.

[5] https://pypi.org/project/bitcoinlib/

The verifier functions in two main phases: (a) **submit phase** and (b) **contest phase**. Each phase has different input and functionality, and is performed by different entities.

Submit phase:

In **submit phase**, an entity submits a proof and an event. We assume that at least one honest full node is aware of the submission. This is also a part of the model of NIPoPoWs, and is a logical assumption as explained in the NIPoPoW paper. In order to claim the occurrence of an event, one must provide a proof and a predicate regarding the underlying event. If r rounds pass, the value of the predicate becomes immutable. The passing of rounds is indicated by the mining of new blocks atop of the block containing the submitted proof. The value of the predicate can change if a different prover successfully contests the submitted proof at some round $r_c < r$.

Contesting phase:

In **contesting phase**, a new proof is submitted. If this proof has a better score, then the predicate is evaluated against the new proof. The contesting proof is considered better only if it is structurally correct and it represents a chain that encapsulates more proof-of- work than the originally submitted proof, as described in the NIPoPoWs paper. In order to contest, one must provide the new proof and the predicate that is claimed to be `true`by the originally submitted proof.

The functionality of a NIPoPoW verifier is the following:

- If an *honest* party submits a proof and no contest occurs, then the *predicate* becomes *true*.
- If an *honest* party submits a proof and it is contested by an *adversary*, then the contest is unsuccessful and the *predicate* remains *true*.
- If an *adversary* submits a proof, then an *honest* prover makes a contest. The contest invalidates the original submission and the *predicate* becomes $false$.
- The scenario in which an *adversary* submits a proof an *honest* does not contest does not take place due to the assumption that at least one honest party observes the traffic of the contract.

2.8.3 Considerations

In order to construct a verifier, a Directed Acyclic Graph (DAG) is maintained in storage. This structure is stored in the form of a hashmap, and is used to host blocks of all different proofs. This process aims to prevent adversarial proofs which are structurally valid but blocks are intentionally skipped. Such a scenario is displayed in Figure 2.5. The DAG is then used to construct the ancestors structure by performing a simple graph search. By iterating ancestors, we can securely determine the value of the predicate.

Reading from and writing to persistent memory are expensive operations in Solidity [42]. A summary of gas costs for storage and memory access if is displayed in Table 2.1. This fact was observed by Giorgos et al. and was recognized as the bottleneck of the application.

Solidity Algorithm

We describe each phase of the previous implementation in Algorithm 4.

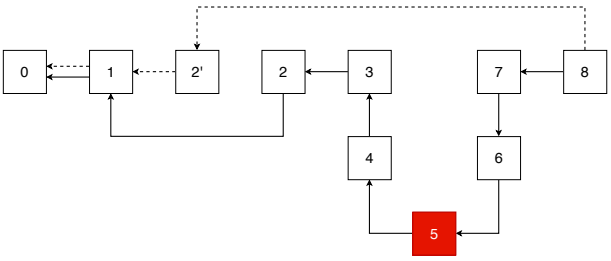


Figure 2.5: Combination of multiple proofs in a DAG. The red block is the block of interest. Honest proof consists of blocks connected by solid lines and adversarial proof by dashed lines. The adversary intentionally uses a different set of blocks.

Operation	Cost	Desc
G_{create}	32000	Paid for a CREATE operation.
G_{sload}	200	Paid for a SLOAD operation.
G_{sset}	20000	Paid for an SSTORE operation.
G_{memory}	3	Paid words expanding memory.
$G_{txdatazero}$	4	Paid for zero byte of data for a transaction.
$G_{txdatanonzero}$	68	Paid for non-zero byte of data for a transaction.

Table 2.1: Ethereum gas list

Algorithm 4 The NIPoPoW verifier implementation by Christoglou et. al.

```
1:  contract crosschain
2:      events ← ⊥
3:      DAG ← ⊥; ancestors ← ⊥
4:      function submit(π_s, e)
5:          require(events[e] = ⊥)
6:          require(valid-interlinks(π))
7:          events[e].π ← π_s
8:          DAG ← DAG ∪ π
9:          ancestors ← find-ancestors(DAG, π[-1])
10:         require(evaluate-predicate(ancestors, e))
11:         ancestors = ⊥
12:     end function
13:     function contest(π_c, e)
14:         require(events[e] ≠ ⊥)
15:         require(valid-interlinks(π_c))
16:         lca = find-lca(events[e].π, π_c)
17:         require(π_c ≥_m events[e].π)
18:         DAG ← DAG ∪ π_c
19:         ancestors ← find-ancestors(DAG, events[e].π[-1])
20:         require(¬evaluate-predicate(ancestors, e))
21:         ancestors = ⊥
22:         events[e] ← ⊥
23:     end function
24: end contract
```

Chapter 3

Set Up and Analysis of Previous Work

3.1 Setting Up a Development Environmnet

The first step towards implementation is to set up a comfortable and adjustable environment. There are several environments one can use to build Solidity applications, most popular of which are Truffle[1], Remix[2] and Embark[3]. However, none of the aforementioned applications delivered the experience we needed in the scope of our project, due to the lack of speed and customization options. That led us to the creation of a custom environment for importing, compiling, deploying and testing smart contracts.

We used Python 3.8[4] to build our environment, since it is a powerful and convenient programming language, and all dependencies we needed had available Python implementations. We developed our environment in Linux[5].

The components we used as building blocks are Web3[6] which is a powerful library for interacting with Ethereum and the Solidity v0.6.10 compiler[7]. For the purpose of our project, we deploy a private blockchain. This is a common practice for Ethereum development since it greatly facilitates testing procedures. Our environment supports multiple EVMs, namely Geth[8], Ganache[9] and Py-EVM[10].

3.1.1 Configuration

We observe that selecting and using an EVM for testing purposes is not trivial. The set of configurations we used for each EVM can be found in our public repository[11]. We hope that this will facilitate future work.

In this chapter, we analyse the work of Christoglou et. al. First we discuss the process needed to prepare the code for our analysis. Then, we show the gas usage of all internal functions of the verifier, and the cost of using this implementation. Finally, we present the vulnerabilities we discovered, and how we mitigated them in our work.

3.1.2 Porting from old Solidity version

We used the latest version of Solidity compiler for our analysis. In order to perform this analysis, we needed to port the verifier from version Solidity 0.4 to version 0.6. The changes we needed to perform were mostly syntactic. These includes the usage of `abi.encodePacked`,

[1] https://trufflesuite.com/
[2] https://remix.ethereum.org/
[3] https://framework.embarklabs.io/
[4] https://python.org/
[5] https://archlinux.org/
[6] https://py.readthedocs.io/
[7] https://solidity.readthedocs.io/
[8] https://geth.ethereum.org/
[9] https://trufflesuite.com/ganache
[10] https://pypi.org/project/py-evm/
[11] https://github.com/sdaveas/setup-geth

explicit `memory` and `storage` declaration and explicit cast from `address` to `payable address`. We also used our configured EVMs with EIP 2028 [15] enabled to benefit from low cost function calls. The functionality of the contract remained equivalent.

3.1.3 Gas analysis

Our profiler measures gas usage per line of code. This is very helpful to observe detailed gas consumption of a contract. Also, we used Solidity events to measure aggregated gas consumption of different high-level functionalities by utilizing the build-in `gasleft()` function. For our experiment, we used a chain of 75 blocks and a forked chain at index 55 that spans 10 additional blocks as displayed in Figure 3.1. Detailed gas usage of the functionalities of the verifier is shown in Table 3.1.

Submit function	**gas usage**
validate Interlink	465,604
store proof	1,044,705
store DAG	3,168,612
find ancestors	4,995,289
evaluate predicate	306,433
delete ancestors	45,137
Sum	10,025,780

Contest function	**gas usage**
validate Interlink	485,751
find LCA	1,255,523
compare proofs	447,130
store proof	304,845
update DAG	1,836,578
find ancestors	5,584,173
evaluate predicate	390,307
delete ancestors	57,023
Sum	10,361,330

Table 3.1: Execution for proof of 75 blocks

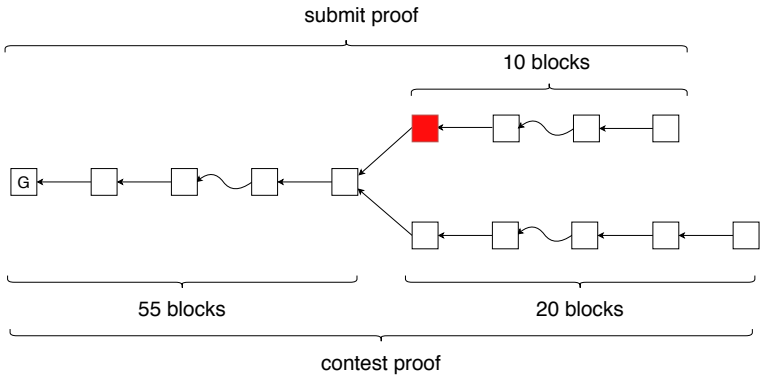


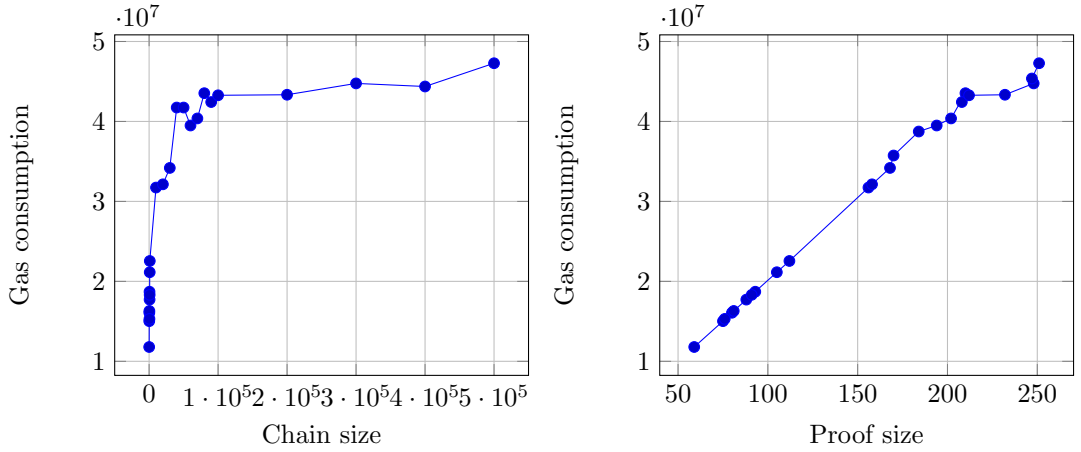
Figure 3.1: The red block indicates the block of interest. Curved connections imply intermediate blocks. The adversary creates a proof for an event that does not exist in the honest chain

In this scenario, the original proof is created by an adversary for an event that does not exist in the honest chain. The proof is contested by an honest party. We select this configuration because it includes both phases (submit and contest) and provides full code coverage of each phase since all underlying operations are executed and no early returns occur.

For a chain of 75 blocks, each phase of the contract needed more than 10 million gas units. Although the size of this test chain is only a very small faction of the size of a realistic chain, the gas usage already exceeds the limit of Ethereum blockchain, which is slightly below 10 million. In particular, the submit of a 650,0000-blocks chain demands 47,280,453 gas. In Figure 3.2, we show gas consumption of the submit phase for different chain sizes and their corresponding

proofs sizes. We demonstrate results for chain sizes from 100 blocks (corresponding to proof size 25) to 650,000 blocks (corresponding to proof size 250).

The linear relation displayed in Figure 3.2b implies that the gas consumption of the verifier is determined by the size of the proofs. As shown in Figure 3.2a, the size of the proofs grows logarithmically to the size of the chain, and this is also reflected to the gas consumption curve.



(a) Gas consumption per chain size

(b) Gas consumption per proof size

Figure 3.2: Gas consumption with respect to chain and corresponding proof size

Pricing So far, we have shown that the verifier is not applicable to the real blockchain due to extensive gas usage, exceeding the build-in limitation the Ethereum blockchain by far. While Ethereum adapts to community demands and build-in parameters can change, it seem improbable to ever incorporate such a high block gas limit. However, even in this extreme case, the verifier would still be impractical due to the high cost of the operations in fiat. We call this amount *toll*, because it is the cost of using the "bridge" between two blockchains. We list these tolls in Table 3.2. For this price, we used gas price equal to 5 Gwei, which is considered a particularly low amount to complete transactions. With this gas price, the probability approximately that the transaction will be included in one of the next 200 blocks is 33%. Note that low and average gas price will not be sufficient for contesting phase, and it has to be performed with higher gas price because of the limited contesting period. We will later analyze thoroughly the entire spectrum of gas prices and tolls for realistic usage of both submit and contest phases.

Chain size	**Toll**
100	12.43 €
500	16.14 €
1,000	23.79 €
10,000	33.47 €
50,000	44.03 €
100,000	45.65 €
500,000	49.88 €

Table 3.2: Tolls for different chain sizes. Gas price is Gwei

3.1.4 Security Analysis

We observed that previous work is vulnerable to *premining attack*. We now lay out an attack, and show how our work mitigates the threat.

Premining

Premining is the creation of a number of cryptocurrency coins before the cryptocurrency is launched to the public. There are altcoins that are based on premining such as AuroraCoin [8]. Bitcoin, however, is *not* a premined cryptocurrency, since it is proven that the genesis was created after 3/Jan/2009. In a blockchain where such a guarantee did not exist, the creator of the chain could quietly mine blocks for a long time before initiating the public network as displayed in Figure 3.3. An adversary could then publish the private, longer chain, invalidating the public chain which is adopted by the honest users and hosts all their funds.

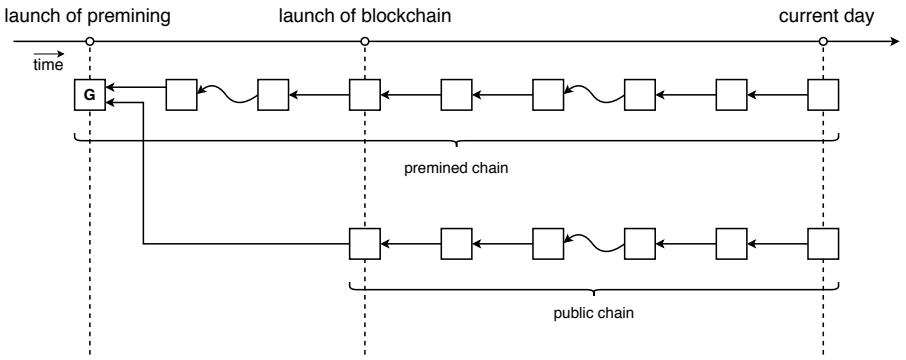


Figure 3.3: A premined chain started before the initiation of the public network. The older chain contains more blocks and proof-of-work.

The NIPoPoW protocol takes into consideration the *genesis* block of the underlying blockchain. We remind that the first block of the chain is always included in the NIPoPoW by construction. In the NIPoPoW protocol a proof is structurally correct if two properties are satisfied:

(a) The *interlink* structure of all proof blocks is valid.

(b) The first block of a proof for a chain C is the first block C, the *genesis* block.

From property (b) of NIPoPoWs, we derive that the protocol is resilient to premining because any chain that does not start from Bitcoin's *genesis* block, $\mathcal{G}$, results to a proof that also does not start from $\mathcal{G}$. Premined chains start with blocks different than $\mathcal{G}$, hence proofs that describe premined chains are invalid by definition.

An attack

Previous implementation does require the existence of underlying chain's *genesis*, exposing the verifier to premining attacks. Such an attack can be launched by an adversary that mines new blocks on a chain C_p which is prior to the Bitcoin chain C as displayed in 3.4.

Proofs for events in C_p cannot be contested by an honest party, since chain C_p includes more proof-of-work than C, and thus proofs with higher score can be generated.

Mitigation

We can mitigate this vulnerability by initializing the smart contract with the *genesis* block of the underlying blockchain (in our case, Bitcoin) persisting *genesis* in storage. For every phase, we add an assertion that the first block of the proof must be equal to *genesis*. The needed changes are shown in Algorithm 5.

These operation do not affect the cost of the verifier because the extra data saved in storage is constant and small in size (32 bytes).

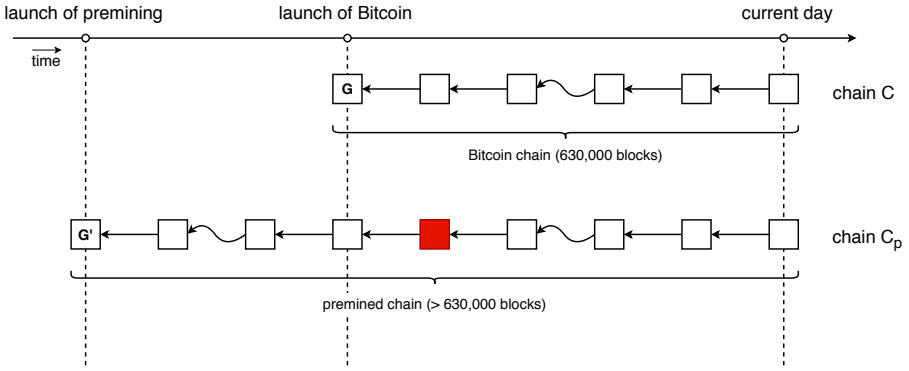


Figure 3.4: A premined chain started before Bitcoin. The older chain contains more blocks.

Algorithm 5 The NIPoPoW client mitigation to premining attack

```
1:  contract crosschain
2:      events ← ⊥; 𝒢 ← ⊥
3:      DAG ← ⊥; ancestors ← ⊥
4:      function initialize(𝒢_remote)
5:          𝒢 ← 𝒢_remote                ▷ initialize with the genesis of the underlying chain
6:      end function
7:      function submit(π_s, e)
8:          require(π_s[0] = 𝒢)                                  ▷ assert correct genesis
9:          require(events[e] = ⊥)
10:         require(valid-interlinks(π))
11:         events[e].π ← π_s
12:         DAG ← DAG ∪ π
13:         ancestors ← find-ancestors(DAG, π[-1])
14:         require(evaluate-predicate(ancestors, e))
15:         ancestors = ⊥
16:     end function
17:     function contest(π_c, e)
18:         require(π_c[0] = 𝒢)                                  ▷ assert correct genesis
19:         require(events[e] ≠ ⊥)
20:         require(valid-interlinks(π_c))
21:         lca = find-lca(events[e].π, π_c)
22:         require(π_c ≥_m events[e].π)
23:         DAG ← DAG ∪ π_c
24:         ancestors ← find-ancestors(DAG, events[e].π[-1])
25:         require(¬evaluate-predicate(ancestors, e))
26:         ancestors = ⊥
27:         events[e] ← ⊥
28:     end function
29: end contract
```

Chapter 4

Implementation

write intro

4.1 Storage vs Memory

We will first demonstrate the difference in gas usage between storage and memory for a smart contract in Solidity. Suppose that we have the following simple contract:

```
pragma solidity ^0.6.6;

contract StorageVsMemory {
    uint256 size;
    uint256[] storageArr;

    constructor(uint256 _size) public {
        size = _size;
    }

    function withStorage() public {
        for (uint256 i = 0; i < size; i++) {
            storageArr.push(i);
        }
    }

    function withMemory() public view {
        uint256[] memory memoryArr = new uint256[](size);
        for (uint256 i = 0; i < size; i++) {
            memoryArray[i] = i;
        }
    }
}
```

Listing 4.1: Solidity test for storage and memory

Function `withStorage()` populates an array saved in storage and function `withMemory()` populates an array saved in memory. We initialize the sizes of the arrays by passing the variable size to the contract constructor. We run this piece of code for `size` from 1 to 100. The results are displayed at Figure 4.1. For size equal to 100, the gas expended is 53,574 gas units using memory and 2,569,848 using storage which is almost 50 times more expensive. This code was compiled with Solidity version 0.6.6 with optimizations enabled[1]. The EVM we used was Ganache at the latest Istanbul[2] fork. It is obvious that, if there is the option to use memory instead of storage in the design of smart contracts, the choice of memory greatly benefits the user.

[1] This version of Solidity compiler, which was the latest at the time this paper was published, did not optimize-out any of the variables.

[2] A summary of Istanbul fork can be found in the following link: https://eth.wiki/roadmap/istanbul

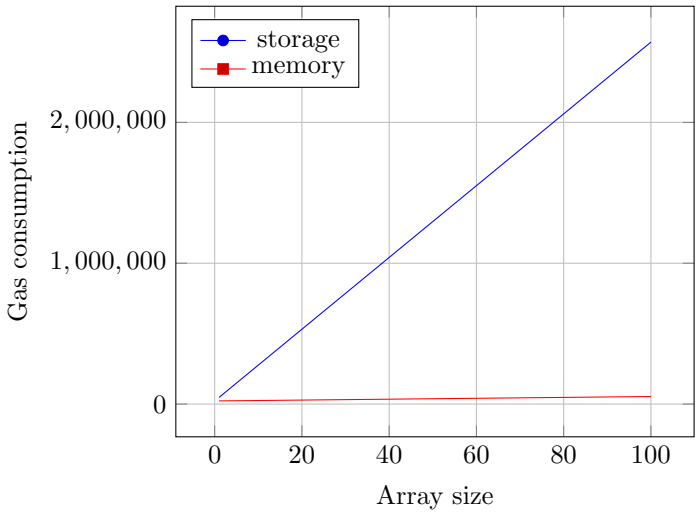


Figure 4.1: Gas consumption for memory and storage

4.2 The Hash-and-Resubmit Pattern

We now introduce a novel design pattern for Solidity smart contracts that results into significant gas optimization due to the elimination of expensive storage operations. We first introduce our pattern, and display how smart contracts benefit from using it. Then, we proceed into integrating our pattern in the NIPoPoW protocol, and we analyze the performance in comparison with previous work [12].

4.2.1 Motivation

It is essential for smart contracts to store data in the blockchain. However, interacting with the storage of a contract is among the most expensive operations of the EVM [42, 5]. Therefore, only necessary data should be stored and redundancy should be avoided when possible. This is contrary to conventional software architecture, where storage is considered cheap. Usually, the performance of data access in traditional systems is related with time. In Ethereum, however, performance is related to gas consumption. Access to persistent data costs a substantial amount of gas, which has a direct monetary value.

One way to mitigate gas cost of reading variables from the blockchain is to declare them as public. This leads to the creation of a *getter* function in the background, allowing free access to the value of the variable. But this treatment does not prevent the initial population of storage data and write operations which are significantly expensive for large sizes of data.

By using the *hash-and-resubmit* pattern, large structures are omitted from storage entirely, and are contained in memory. When a function call is performed, the signature and arguments of the function are included in the transactions field of the body of a block. The contents of blocks are public to the network, therefore this information is locally available to full nodes. By simply observing blocks, a node retrieves data sent to the contract by other users. To interact publicly with this data without the utilization storage, the node *resends* the observed data to the blockchain. The concept of resending data is redundant in conventional systems. However, this technique is very efficient to use in Solidity due to the significantly lower gas cost of memory operations in relation with storage operations.

4.2.2 Related patterns

Towards implementing gas-efficient smart contracts [6, 7, 17, 21], several methodologies have been proposed. In order to eliminate storage operations using data signatures, the utilization

of IPFS [3] is proposed by [37] and [22]. However, these solutions do not address availability, which is one of our main requirements. Furthermore, [13] uses logs to replace storage in a similar manner, sparing a great amount of gas. However, this approach does not address consistency, which is also one of our critical targets. Lastly, [41] proposes an efficient manner to replace read storage operations, but does not address write operations.

4.2.3 Applicability

We now list the requirements an application needs to meet in order to benefit from the *hash-and-resubmit* pattern:

1. The application is a Solidity smart contract.
2. Read/write operations are performed in large arrays that exist in storage. Using the pattern for variables of small size may result in negligible gain or even performance loss.
3. A full node observes function calls to the smart contract.

4.2.4 Participants and collaborators

The first participant is the smart contract S that accepts function calls. Another participant is the invoker E_1, who dispatches a large array d_0 to S via a function $\mathsf{func}_1(\mathsf{d}_0)$. Note that d_0 is potentially processed in func_1, resulting to d. The last participant is the observer E_2, who is a full node that observes transactions towards S in the blockchain. This is possible because nodes maintain the blockchain locally. After observation, E_2 retrieves data d. Since this is an off-chain operation, a malicious E_2 potentially alters d before interacting with S. We denote the potentially modified d as d^*. Finally, E_2 acts as an invoker by making a new call to S, $\mathsf{func}_2(\mathsf{d}^*)$. The verification that $\mathsf{d} = \mathsf{d}^*$, which is a prerequisite for the secure functionality of the underlying contract, consists a part of the pattern and is performed in func_2.

4.2.5 Implementation

The implementation of this pattern is divided in two parts. The first part covers how d^* is retrieved by E_2, whereas in the second part the verification of $\mathsf{d} = \mathsf{d}^*$ is realized. The challenge here is twofold:

1. Availability: E_2 must be able to retrieve d without the need of accessing on-chain data.
2. Consistency: E_2 must be prevented from dispatching d^* that differs from d which is a product of originally submitted d_0.

Hash-and-resubmit technique is performed in two stages to face these challenges: (a) the *hash* phase, which addresses *consistency*, and (b) the *resubmit* phase which addresses *availability* and *consistency*.

Addressing Availability

During the *hash* phase, E_1 makes the function call $\mathsf{func}_1(\mathsf{d}_0)$. This transaction, which includes a function signature (func_1) and the corresponding data (d_0), is added in a block by a miner. Due to blockchain's transparency, the observer of func_1, E_2, retrieves a copy of d_0 from the calldata, without the need of accessing contract data. In turn, E_2 performs *locally* the same set of on-chain instructions operated on d_0, generating d. Thus, availability is addressed through observability.

Addressing Consistency

We prevent an adversary E_2 from dispatching data $d^* \neq d$ by storing the *signature* of d in the contract's state during the execution of $func_1(.)$ by E_1. In the context of Solidity, a signature of a structure is the digest of the structure's *hash*. The pre-compiled `sha256` is convenient to use in Solidity, however we can make use of any cryptographic hash function $H()$:

$$hash \leftarrow H(d)$$

Then, in *rehash* phase, the verification is performed by comparing the stored digest of d with the digest of d^*:

$$require(hash = H(d^*))$$

In Solidity, the size of this digest is 32 bytes. To persist such a small value in the contract's memory only adds a constant, negligible cost overhead. We illustrate the application of the *hash-and-resubmit* pattern in Figure 4.2.

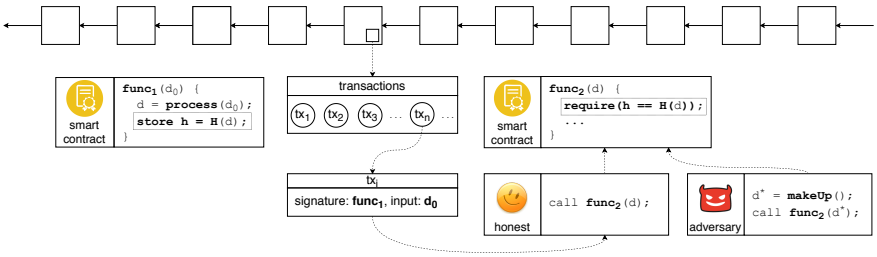


Figure 4.2: The *hash-and-resubmit* pattern. First, an invoker calls $func_1(d_0)$. d_0 is processed *on-chain* and d is generated. The signature of d is stored in the blockchain as the digest of a hash function $H(.)$. Then, a full node that observes invocations of $func_1$ retrieves d_0, and generates d by performing the analogous processing on d_0 *off-chain*. An adversarial observer dispatches d^*, where $d^* \neq d$. Finally, the nodes invoke $func_2(.)$. In $func_2$, the validation of input data is performed, reverting the function call if the signatures of the input does not match with the signature of the originally processed data. By applying a *hash-and-resubmit pattern*, only the fixed-size signature of d is stored to the contract's state, replacing arbitrarily large structures.

4.2.6 Sample

We now demonstrate the usage of the hash-and-resubmit pattern with a simplistic example. We create a smart contract that orchestrates a game between two players, P_1 and P_2. The winner is the player with the most valuable array. The interaction between players through the smart contract is realized in two phases: (a) the submit phase and (b) the contest phase.
Submit phase: P_1 submits an N-sized array, a_1, and becomes the holder of the contract.
Contest phase: P_2 submits a_2. If the result of $compare(a_2, a_1)$ is true, then P_2 becomes the holder.

We provide a simple implementation for compare, but we can consider any notion of comparison, since the pattern is abstracted from such implementation details.

We make use of the *hash-and-resubmit* pattern by prompting P_2 to provide *two* arrays to the contract during contest phase: (a) a_1^*, which is the originally submitted array by P_1, possibly modified by P_2, and (b) a_2, which is the contesting array.

We provide two implementations of the above described game. In Listing 4.2 we display the storage implementation (left), and the memory implementation (right) embedding the *hash-and-resubmit* pattern.

```
1 // Storage implementation                    | // Memory implementation
2                                               |
```

```
pragma solidity ^0.6.0;                          | pragma solidity ^0.6.0;
                                                 |
contract StorageGame {                           | contract MemoryGame {
  uint256[] public a1;                           +   bytes32 public commit;
  address public holder;                         |   address public holder;
                                                 |
  function submit(uint256[] memory a)            |   function submit(uint256[] memory
      a)                                         |
  public                                         |   public
  {                                              |   {
    // Save array in storage                     +     // Same hash of array
    a1 = a;                                      +     commit = sha256(abi.
        encodePacked(a));                        |
    holder = msg.sender;                         |     holder = msg.sender;
  }                                              |   }
                                                 |
  // Pass contesting array                       +   // Pass original and contesting
      array                                      |
  function contest(uint256[] memory a)           +   function contest(uint256[] memory
      a1,                                        |
                                                 +                     uint256[] memory
                                                 |    a2)
  public                                         |   public
  {                                              |   {
                                                 +     // Check commitment of original
                                                 |    array
                                                 +     require(sha256(abi.encodePacked
                                                 |   (a1)) ==
                                                 +              commit);
    require(compare(a));                         +     require(compare(a1, a2));
    holder = msg.sender;                         |     holder = msg.sender;
  }                                              |   }
                                                 |
  // Compare with storage array                  |   // Compare with memory array
  function compare(uint256[] memory a2)          +   function compare(uint256[] memory
      a1,                                        |
                                                 +                     uint256[] memory
                                                 |    a2)
  internal view returns(bool)                    +   internal pure returns(bool)
  {                                              |   {
    if (a2.length < a1.length) {                 |     if (a2.length < a1.length) {
      return false;                              |       return false;
    }                                            |     }
    for (uint i = 0; i < a1.length; i++) {       |     for (uint i = 0; i < a1.length;
        i++) {                                   |
      if (a2[i] < a1[i]) {                       |       if (a2[i] < a1[i]) {
        return false;                            |         return false;
      }                                          |       }
    }                                            |     }
    return true;                                 |     return true;
  }                                              |   }
}                                                | }
```

Listing 4.2: Implementation of sample.

4.2.7 Gas analysis

The gas consumption of the two above implementations is displayed in Figure 4.3. By using the *hash-and-resubmit* pattern, the aggregated gas consumption for submit and contest is decreased by 95%. This significantly affects the efficiency and applicability of the contract. Note that the storage implementation exceeds the Ethereum block gas limit (10,000,000 gas as of June 2020), for arrays of size 500 and above, contrary to the optimized version, which consumes approximately only $1/10^{th}$ of the block gas limit for arrays of 1,000 elements.

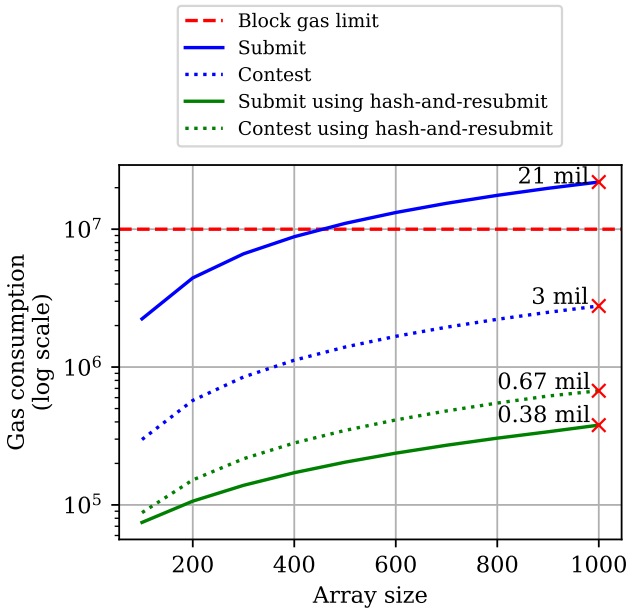


Figure 4.3: Gas-cost reduction using the *hash-and-resubmit* pattern (lower is better). By avoiding gas-heavy storage operations, the aggregated cost of submit and contest is decreased by 95%.

4.2.8 Consequences

The most obvious consequence of applying the *hash-and-resubmit* pattern is the circumvention of storage structures, a benefit that saves a substantial amount of gas, especially when these structures are large. To that extent, smart contracts that exceed the Ethereum block gas limit and benefit sufficiently for the alleviation of storage structures are becoming practical. Furthermore, the pattern enables off-chain transactions, significantly improving the performance of smart contracts.

4.2.9 Known Uses

To our knowledge, we are the first to address consistency and availability by combining blockchain's transparency with data structures signatures in a manner that eliminates storage variables from smart contracts.

4.2.10 Enabling NIPoPoWs

We now present how the *hash-and-resubmit* pattern is used in the context of the NIPoPoW superlight client. The NIPoPoW verifier adheres to a submit-and-contest schema where the inputs of the functions are arrays that are processed on-chain, and a node observes function calls towards the smart contract. Therefore, it makes a suitable case for our pattern.

In the *submit* phase, a *proof* is submitted. In the case of falsity, it is contested by another user in *contest* phase. The contester is a node that monitors the traffic of the verifier. The input of submit function includes the submit proof (π_s) that indicates the occurrence of an *event* (e) in the source chain, and the input of contest function includes a contesting proof (π_c). A successful contest of π_s is realized when π_c has a better score [27]. In this section, we will not examine the score evaluation process since it is irrelevant to the pattern. The size of proofs is dictated by the value m. We consider $m = 15$ to be sufficiently secure [27].

In previous work, NIPoPoW proofs are maintained on-chain, resulting in extensive storage operations that limit the applicability of the verifier considerably. In our implementation, proofs are not stored on-chain, and π_s is retrieved by the contester from the calldata. Since we assume a trustless network, π_s is altered by an adversarial contester. We denote the potentially changed π_s as π_s^*. In *contest* phase, π_s^* and π_c are dispatched in order to enable the *hash-and-resubmit* pattern.

For our analysis, we create a blockchain similar to the Bitcoin chain with the addition of the interlink structure in each block as in [12]. Our chain spans 650,000 blocks, representing a slightly enhanced Bitcoin chain[3]. From the tip of our chain, we branch two sub-chains that span 100 and 200 additional blocks respectively, as illustrated in Figure 4.4. Then, we use the smaller chain to create π_s, and the larger chain to create π_c. We apply the protocol by submitting π_s, and contesting with π_c. The contest is successful, since π_c represents a chain consisting of greater number of blocks than π_s, therefore encapsulating more proof-of-work. We select this setting as it provides maximum code coverage, and it describes the most gas-heavy scenario for the verifier.

In Algorithm 6 we show how *hash-and-resubmit* pattern is embedded into the NIPoPoW client.

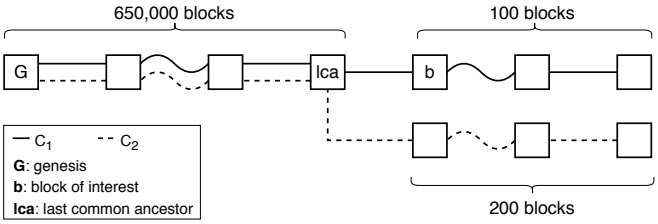


Figure 4.4: Forked chains for our gas analysis.

In Figure 4.5, we display how the *hash-and-resubmit* provides an improved implementation compared to previous work. The graph illustrates the aggregated cost of *submit* and *contest* phases for each implementation. We observe that, by using the *hash-and-resubmit* pattern, we achieve to increase the performance of the contract by 50%. This is a decisive step towards creating a practical superlight client.

Note that gas consumption generally follows an ascending trend, however it is not a monotonically increasing function. This is due to the fact that NIPoPoWs are probabilistic structures, the size of which is determined by the distribution of superblocks within the underlying chain. A proof that is constructed for a chain of a certain size can be larger than a proof constructed for a slightly smaller chain, resulting in non-monotonic increase of gas consumption between consecutive values of proof sizes.

[3] Bitcoin spans 633,022 blocks as of June 2020. Metrics by https://blockchain.com/

Algorithm 6 The NIPoPoW client using hash-and-resubmit pattern

```
1: contract crosschain
2:     events ← ⊥; G ← ⊥
3:     function initialize(G_remote)
4:         G ← G_remote
5:     end function
6:     function submit(π_s, e)
7:         require(events[e] = ⊥)
8:         require(π_s[0] = G)
9:         require(valid-interlinks(π))
10:        DAG ← DAG ∪ π_s
11:        ancestors ← find-ancestors(DAG, π_s[-1])
12:        require(evaluate-predicate(ancestors, e))
13:        ancestors = ⊥
14:        events[e].hash ← H(π_s)        ▷ enable pattern by storing the hash of π_s instead of π_s
15:    end function
16:    function contest(π*_s, π_c, e)                                   ▷ provide proofs
17:        require(events[e] ≠ ⊥)
18:        require(events[e].hash = H(π*_s))                            ▷ verify π*_s
19:        require(π_c[0] = G)
20:        require(valid-interlinks(π_cont))
21:        lca = find-lca(π*_s, π_c)
22:        require(π_c ≥_m π*_s)
23:        DAG ← DAG ∪ π_c
24:        ancestors ← find-ancestors(DAG, π*_s[-1])
25:        require(¬evaluate-predicate(ancestors, e))
26:        ancestors = ⊥
27:        events[e] ← ⊥
28:    end function
29: end contract
```

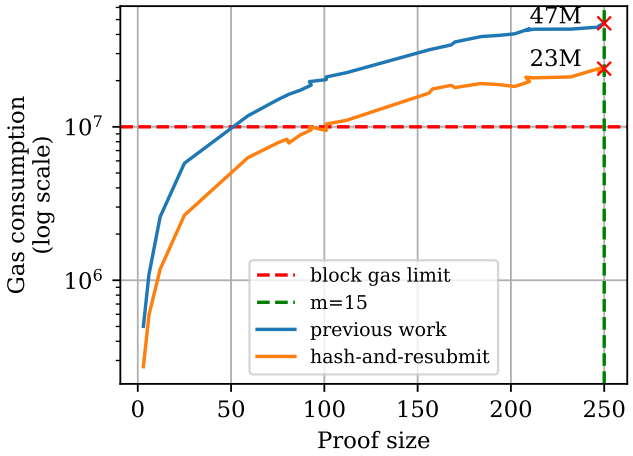


Figure 4.5: Performance improvement using hash-and-resubmit pattern in NIPoPoWs compared to previous work for a secure value of m (lower is better). The gas consumption is decreased by approximately 50%.

4.3 Hash-and-Resubmit Variations

In order to enable selective dispatch of a segment of interest, different hashing schemas can be adopted, such as Merkle Trees [32] and Merkle Mountain Ranges [29, 40]. In this variation of the pattern, which we term *merkle-hash-and-resubmit*, the signature of an array d is Merkle Tree Root (MTR). In the *resubmit* phase, d[m:n] is dispatched, accompanied by the siblings that reconstruct the MTR of d.

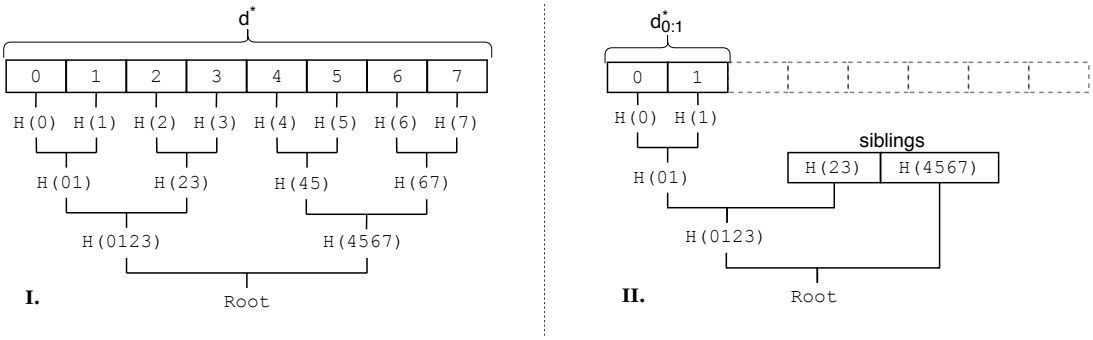


Figure 4.6: **I.** The calculation of root in *hash* phase. **II.** The verification of the root in *resubmit* phase. H(k) denotes the digest of element k. H(kl) denotes the result of H(H(k) | H(l))

This variation of the pattern removes the burden of sending redundant data, however it implies on-chain construction and validation of the Merkle construction. In order to construct a MTR for an array d, |d| hashes are needed for the leafs of the MT, and |d| − 1 hashes are needed for the intermediate nodes. For the verification, the segment of interest d[m:n] and the siblings of the MT are hashed. The size of siblings is approximately log_2(|d|). The process of constructing and verifying the MTR is displayed in Figure 4.6.

In Solidity, different hashing operations vary in cost. An invocation of sha256(d), copies d in memory, and then the *CALL* instruction is performed by the EVM that calls a precompiled contract. At the current state of the EVM, *CALL* costs 700 gas units, and the gas

paid for every word when expanding memory is 3 gas units [42]. Consequently, the expression $1 \times \mathsf{sha256}(\mathsf{d})$ costs less than $|\mathsf{d}| \times \mathsf{sha256}(1)$ operations. A different cost policy applies for keccak [4] hash function, where hashing costs 30 gas units plus 6 additional gas far each word for input data [42]. The usage of keccak dramatically increases the performance in comparison with sha256, and performs better than plain rehashing if the product of on-chain processing is sufficiently larger than the originally dispatched data. Costs of all related operations are listed in Table 4.1.

Operation	Gas cost
load(d)	$\mathsf{d}_{bytes} \times 68$
sha256(d)	$\mathsf{d}_{words} \times 3 + 700$
keccak(d)	$\mathsf{d}_{words} \times 6 + 30$

Table 4.1: Gas cost of EVM operations as of June 2020.

The merkle variation can be potentially improved by dividing d in chunks larger than 1 element. We leave this analysis for future work.

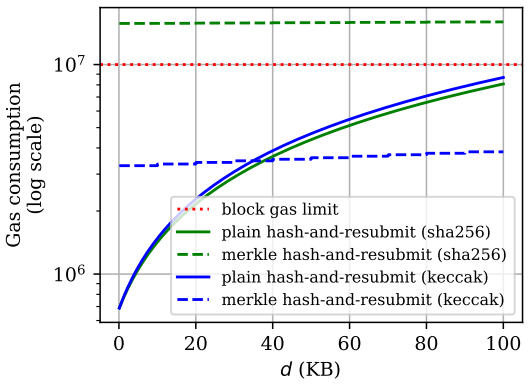


Figure 4.7: Trade-offs between *hash-and-resubmit* variations. In the vertical axis the gas consumption is displayed. In the horizontal axis the size of d. The size of d_0 is 10KB bytes, and the hash functions we use are the pre-compiled `sha256` and `keccak`.

In Table 4.2 we display the operations needed for hashing and verifying the underlying data for both variations of the pattern as a function of data size. In Figure 4.7 we demonstrate the gas consumption for dispatched data of 10KB, and varying size of on-chain process product.

phase per variance	plain hash and resubmit	merkle hash and resubmit
hash	H(d)	$\mathsf{H}(\mathsf{d}_{elem}) \times \lvert\mathsf{d}\rvert$ $\mathsf{H}(\text{digest}) \times (\lvert\mathsf{d}\rvert - 1)$
resubmit	load(d) + H(d)	load(d[m:n]) + load($siblings$) + H(d[m:n]) + H($digest$)$\times\lvert siblings\rvert$

Table 4.2: Summary of operations for *hash-and-resubmit* pattern variations. d is the product of on-chain operations and d_{elem} is an element of d. H is a hash function, such as sha256 or keccak, *digest* is the product of H(.) and *siblings* are the siblings of the Merkle Tree constructed for d.

4.4 Removing Look-up Structures

Now that we freely eliminate large arrays, we can focus on other types of storage variables. The challenge we face is that the protocol of NIPoPoWs depends on a Directed Acyclic Graph (DAG) of blocks which is a mutable hashmap. This DAG is needed because interlinks of superblocks can be adversarially defined. By using DAG, the set of ancestor blocks of a block is extracted by performing a simple graph search. For the evaluation of the predicate, the set of *ancestors* of the best blockchain tip is used. Ancestors are created to avoid an adversary who presents an honest chain but skips the blocks of interest.

This logic is intuitive and efficient to implement in most traditional programming languages such as C++, JAVA, Python, JavaScript, etc. However, as our analysis demonstrates, such an implementation in Solidity is significantly expensive. Albeit Solidity supports constant-time look-up structures, hashmaps are only contained in storage. This affects the performance of the client, especially for large proofs.

We make a keen observation regarding potential positions of the *block of interest* in proofs, which leads us to the construction of an architecture that does not require a DAG, the ancestors or other complementary structures. To support this claim, we adopt the notation from [27]. We also consider the predicate p to be of the type: "does block B exist inside proof π?", where B denotes the block of interest of proof π. The entity that performs the submission is $\mathsf{E_s}$, and the entity that initiates a contest is $\mathsf{E_c}$.

4.4.1 Position of Block of Interest

NIPoPoWs are sets of sampled interlinked blocks, meaning that they can be perceived as chains. Since proofs π_s and π_c differ, a fork is created at the index of their last common ancestor (LCA). The block of interest lies at a certain index within π_s and indicates a stable predicate [27, 30] that is true for π_s. A submission in which B is absent from π_s is aimless, because it automatically fails since no element of π_s satisfies p. On the contrary, π_c tries to prove the *falseness* of the underlying predicate. This means that, if the block of interest is included in π_c, then the contest is aimless. We freely use the term aimless to also characterize components that are included in such actions i.e. aimless proof, aimless blocks etc. We use the term meaningful to describe non-aimless actions and components.

In the NIPoPoW protocol, proofs' segments $\pi_\mathsf{s}\{:\mathsf{LCA}\}$ and $\pi_\mathsf{c}\{:\mathsf{LCA}\}$ are merged to prevent adversaries from skipping blocks, and the predicate is evaluated against $\pi_\mathsf{s}\{:\mathsf{LCA}\} \cup \pi_\mathsf{c}\{:\mathsf{LCA}\}$. We observe that $\pi_\mathsf{c}\{:\mathsf{LCA}\}$ can be omitted, because no block B exists such that $\{\mathsf{B} : \mathsf{B} \notin \pi_\mathsf{s}\{:\mathsf{LCA}\} \wedge \mathsf{B} \in \pi_\mathsf{c}\{:\mathsf{LCA}\}\}$ where B results into positive evaluation of the predicate. This is due to the fact that, in a meaningful contest, B is not included in π_c. Consequently, π_c is only meaningful if it forks π_s at a block that is prior to B.

In Figure 4.8 we display a fork of two proofs. Solid lines connect blocks of π_s and dashed lines connect blocks of π_c. By examining which scenarios are meaningful based on different positions of the block of interest, we observe that blocks `B`, `C` and `E` do not qualify, because they are included in π_c. Block `A` is included in $\pi_\mathsf{s}\{:\mathsf{LCA}\}$, which means that π_c is an aimless contest because the LCA comes after the block of interest. Therefore, A is an aimless block as a component of an aimless contest. Given this configuration, the only meaningful block of interest is `D` and its predecessors (which we leave out from this figure).

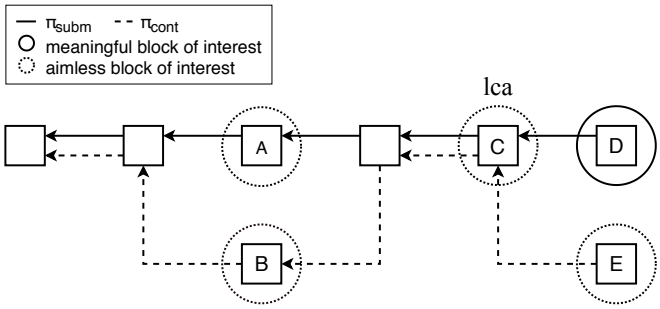


Figure 4.8: Fork of two proofs. Solid lines connect blocks of π_s, and dashed lines connect blocks of π_c. In this configuration, blocks in dashed circles are aimless blocks of interest, and the block in the solid circle is a meaningful block of interest. Blocks B, C and E are aimless because they exist in π_c. Block A is aimless because it belongs to the subchain $\pi_s\{:\mathsf{LCA}\}$

4.4.2 Subset of Proofs

Now that we have achieved to freely retrieve π_s, we can start sketching methodologies that benefit from this schema but another challenge we had to face is that the protocol of NIPoPoWs dependents on DAG which is a hashmap data structure

Our first realization was that instead of creating a DAG of π_s and π_c, we can rather require

$$\pi_s\{: \text{LCA}\} \subseteq \pi_c\{: \text{LCA}\}$$

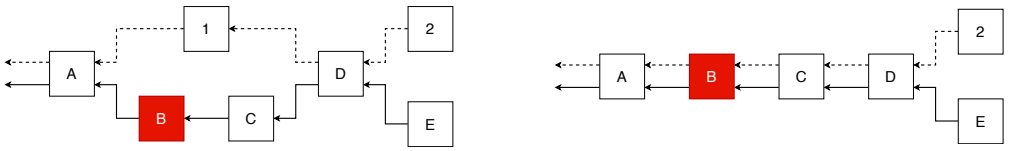
This way, we avoid the burden of maintaining auxiliary structures DAG and ancestors on-chain. The implementation of `subset` is displayed in listing 4.3. The complexity of the function is

$$\mathcal{O}(|\pi_s\{: LCA\}| + |\pi_c\{: LCA\}]|)$$

```
function subset(
    Proof memory exist, uint existLca,
    Proof memory cont, uint contLca
) internal pure returns(bool)
{
    uint256 j = contLca;
    for (uint256 i = existLca; i < exist.length; i++) {
        while (exist[i] != cont[j]) {
            if (++j >= contLca) { return false; }
        }
    }
    return true;
}
```

Listing 4.3: Implementation of subset

The gas consumption difference between *subset* and $DAG + ancestors$ is displayed at figure 4.10. *Subset* solution is approximately 2.7 times more efficient.



(a) Valid π_{cont} DAG+ancestors

(b) Valid π_{cont} using subset

Figure 4.9: The red block is the block of interest. The blocks connected with solid lines indicate π_{exist} and blocks connected with dashed lines indicate π_{cont}. In (a), an adversary can dispatch a flawed proof that skips the block of interest. π_{exits} and π_{cont} are aggregated in the DAG, which is traversed to discover best proof. In (b), the proofs are linearly iterated to determine if $\pi_{exist}\{:\text{LCA}\} \subseteq \pi_{cont}\{:\text{LCA}\}$

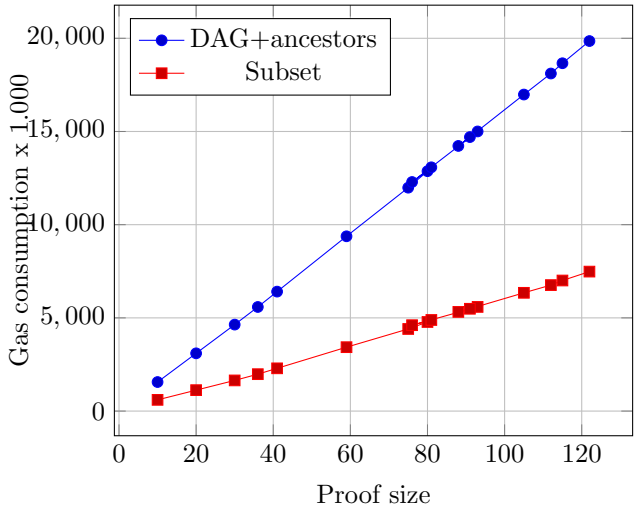


Figure 4.10: Gas consumption for DAG+ancestors and subset

Subset complexity and limitations

Requiring π_s to be a subset of π_c greatly reduces gas, but the complexity of the *subset* algorithm is high since both proofs have to be iterated from *Gen* until lca_e and lca_c, respectively. Generally, we expect from an adversary to provide a proof of a chain that is a fork of the honest chain at some point relatively close to the tip. This is due to the fact that the ability of an adversary to sustain a fork chain exponentially weakens as the honest chain progresses. This means that the length of π, $|\pi|$ is expected to be considerably close to $|\pi\{:lca\}|$, and thus the complexity of `subset()` effectively becomes $\mathcal{O}(2|\pi|)$.

In realistic cases, where LCA lies around index 250 of the proof, the gas cost of `subset()` is approximately 20,000,000 gas units, which makes it inapplicable for real chains since it exceeds the block gas limit of the Ethereum blockchain by far.

4.4.3 Minimal Fork

By combining the above observations, we derive that π_c can be truncated into $\pi_c\{\text{LCA}:\}$ without affecting the correctness of the protocol. We term this truncated proof π_c^f. Security is preserved by requiring π_c^f to be a *minimal fork* of π_s. A minimal fork is a fork chain that shares exactly one common block with the main chain. A proof $\tilde{\pi}$, which is minimal fork of π, has the following attributes:

1. $\pi\{\text{LCA}\} = \tilde{\pi}[0]$
2. $\pi\{\text{LCA}:\} \cap \tilde{\pi}[1:] = \emptyset$

By requiring π_c^f to be a minimal fork of π_s, we prevent adversaries from dispatching an augmented π_c^f to claim better score against π_s. Such an attempt is displayed in Figure 4.11.

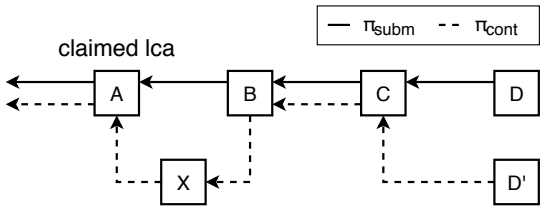


Figure 4.11: An adversary attests to contest with a malformed proof. Adversary proof consists of blocks {A, X, B, C, D'} that achieve better score against submit proof {A, B, C, D}. This attempt is rejected due to the minimal-fork requirement.

The implementation of minimal-fork is shown in listing 4.4. The complexity of `minimalFork()` is:

$$\mathcal{O}(|\pi_s\{\mathsf{LCA}:\}| \times |\pi_c^f|)$$

```
function minimalFork(
    Proof memory exist, uint256 lca
    Proof memory cont
) internal pure returns (bool) {
    for (uint256 i = lca+1; i < exist.length; i++) {
        for (uint256 j = 1; j < contest.length; j++) {
            if (exist[i] == contest[j]) { return false; }
        }
    }
    return true;
}
```

Listing 4.4: Implementation for minimal fork

In Algorithm 7, we show how the minimal fork technique is incorporated into our client replacing DAG and ancestors. In Figure 4.12 we show how the performance of the client improves. We use the same test case as in *hash-and-resubmit*.

By applying the minimal-fork technique, he achieve a 55% decrease in gas consumption. *Submit* phase now costs 4,700,000 gas, and the *contest* phase costs 4,900,000 million gas. This is a notable result, since each phase now fits inside an Ethereum block.

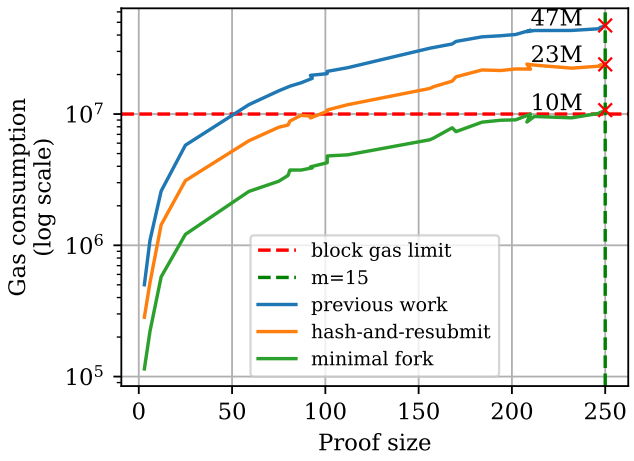


Figure 4.12: Performance improvement using minimal fork (lower is better). The gas consumption is decreased by approximately 55%.

Algorithm 7 The NIPoPoW client using the minimal fork technique

```
1: contract crosschain
2:     ...
3:     function submit(π_s, e)
4:         require(π_s[0] = G)
5:         require(events[e] = ⊥)
6:         require(valid-interlinks(π_s))
7:         require(evaluate-predicate(π_s, e))
8:         events[e].hash ← H(π_s)
9:     end function
10:    function contest(π*_s, π^f_c, e, f)          ▷ f: The index of the fork in π_s
11:        require(events[e] ≠ ⊥)
12:        require(events[e].hash = H(π*_s))
13:        require(valid-interlinks(π^f_c))
14:        require(minimal-fork(π*_s, π^f_c, f))    ▷ Assert that π^f_c is a minimal fork of π*_s
15:        require(π^f_c ≥_m π*_s)
16:        require(¬evaluate-predicate(π^f_c, e))
17:        events[e] ← ⊥
18:    end function
19:    function minimal-fork(π_1, π_2, f)
20:        if π_1[f] ≠ π_2[0] then                  ▷ Check the fork head
21:            return false
22:        end if
23:        for b_1 in π_1[f+1:] do                  ▷ Check if proofs are disjoint
24:            if b_2 in π_2[1:] then
25:                return false
26:            end if
27:        end for
28:        return true
29:    end function
30: end contract
```

4.5 Processing Fewer Blocks

The complexity of most demanding on-chain operations of the verifier are linear to the size of the proof. This includes the proof validation and the evaluation of score. We now present two techniques that allow for equivalent operations of constant complexity.

4.5.1 Optimistic Schemes

In smart contracts, in order to ensure that users comply with the underlying application's rules, certain actions need to be performed on-chain, e.g. verification of data, balance checks, etc. In a different approach, actions that deviate from the protocol are reverted after honest users indicate them, not allowing diverging entities to gain advantages. Such applications that do not check the validity of actions by default, but rather depend on the intervention of honest users are characterized "optimistic". In the Ethereum community, several projects [31, 36, 20, 18] have emerged that incorporate the notion of optimistic interactions. We observe that such a schema can be embedded into the NIPoPoW protocol, resulting in significant performance gain.

We discussed how the verification in the NIPoPoW protocol is realized in two phases. In *submit* phase, the verification of the π_s is performed. This is necessary in order to prevent adversaries from injecting blocks that do not belong to the chain, or changing existing blocks. A proof is valid for submission if it is *structurally correct*. Correctly structured NIPoPoWs have the following requirements: (a) the first block of the proof is the genesis block of the underlying blockchain and (b) every block has a valid interlink.

Asserting the existence of genesis in the first index of a proof is an inexpensive operation of constant complexity. However, confirming the interlink correctness of all blocks is a process of linear complexity to the size of the proof. Albeit the verification is performed in memory, sufficiently large proofs result into costly submissions since their validation consist the most demanding function of the *submit* phase. In Table 4.3 we display the cost of valid-interlink function which determines the structural correctness of a proof in comparison with the overall gas used in submit.

Process	Gas cost	Total %
verify-interlink	2,200,000	53%
submit	4,700,000	100%

Table 4.3: Gas usage of function verify-interlink compared to overall gas consumption of submit.

4.5.2 Dispute Phase

We observe that the addition of a phase in our protocol alleviates the burden of verifying all elements of the proof by enabling the indication of an individual incorrect block. This phase, which we term *dispute* phase, leverages selective verification of the submitted proof at a certain index. As a constant operation, this significantly reduces the gas cost of the verification process.

In the NIPoPoW protocol, when a proof π_s is submitted by $\mathsf{E_s}$, it is retrieved by a node $\mathsf{E_c}$ from the calldata and the proof is checked for its validity *off-chain*. We observe that, in order to prove a structurally invalid π_s, $\mathsf{E_c}$ only needs to indicate the index in which π_s fails the interlink verification. In the protocol that incorporates *dispute* phase, $\mathsf{E_c}$ calls dispute(π_s^*, i) for a structurally incorrect proof, where i indicates the disputing index of π_s^*. Therefore, only one block is interpreted *on-chain* rather than the entire span of π_s^*.

Note that this additional phase does not imply increased rounds of interactions between $\mathsf{E_s}$ and $\mathsf{E_c}$. If π_s is invalidated in *dispute* phase, then *contest* phase is skipped. Similarly, if π_s is structurally correct, but represents a dishonest chain, then $\mathsf{E_c}$ proceeds directly to *contest* phase without the invocation of *dispute*.

	Phase	Gas
	submit	4.7
	contest	4.9
I.	**Total**	**9.6**

	Phase	Gas	Phase	Gas
	submit	2.2	submit	2.2
	dispute	1.3	contest	4.9
II.	**Total**	**3.5**	**Total**	**7.1**

Table 4.4: Performance per phase. Gas units are displayed in millions. **I**: Gas consumption prior to dispute phase incorporation. **II**: Gas consumption for two independent sets of interactions submit/dispute and submit/contest.

In Table 4.4 we display the gas consumption for two independent cycles of interactions:

1. *Submit* and *dispute* for is structurally incorrect π_{s}.
2. *Submit* and *contest* for structurally correct π_{s} that represents a dishonest chain.

In Algorithm 8, we show the implementation of the *dispute* phase. The integration of *dispute* phase leaves contest unchanged.

4.5.3 Isolating the Best Level

As we discussed, *dispute* and *contest* phases are mutually exclusive. Unfortunately, the same constant-time verification as in the *dispute* phase cannot be applied in a contest without increasing the rounds of interactions for the users. However, we derive a major optimization for the *contest* phase by observing the process of score evaluation.

In NIPoPoWs, after the last common ancestor is found, each proof fork is evaluated in terms of proof-of-work score. Each level encapsulates a different score of proof-of-work, and the level with the best score is representative of the underlying proof. Since the common blocks of the two proofs naturally gather the same score, only the disjoint portions need to be addressed. Consequently, the position of the LCA determines the span of the proofs that will be included in the score evaluation process. Furthermore, it is impossible to determine the score of a proof in the *submit* phase because the position of LCA is yet unknown.

After π_{s} is retrieved from the calldata, the score of both proofs is calculated. This means that the level in which each proof encapsulates the most proof-of-work for each proof is known to $\mathsf{E_c}$. In the light of this observation, $\mathsf{E_c}$ only submits the blocks which consist the *best level* of π_{c}. The number of these blocks is constant, as it is determined by the security parameter m, which is irrelevant to the size of the underlying blockchain. We illustrate the blocks that participate in the formulation of a proof's score and the best level of contesting proof in Figure 4.13.

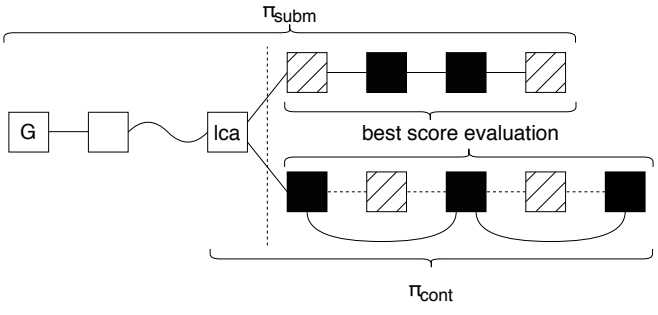


Figure 4.13: Fork of two proofs. Striped blocks determine the score of each proof. Black blocks belong to the level that has the best score. Only black blocks are part of the best level of the contesting proof.

The calculation of the best level of π_{c} is an *off-chain* process. An adversarial $\mathsf{E_c}$ is certainly able to dispatch a level of π_{c} which is different than the best level. However, this is an

irrational action, since different levels only undermine the score of π_c. On the contrary, due to the consistency property of *hash-and-resubmit*, π_s cannot be altered. We denote the best level of π_c^f as $\pi_c^{f,\uparrow^b}$.

In Algorithm 8, we show the implementation of the *contest* phase under the best-level enhancement. The utilization of this methodology greatly increases the performance of the client, because the complexity of the majority of contest functions is related to the size of π_c. In Table 4.5, we demonstrate the difference in gas consumption in the *contest* phase after using *best-level*. The performance of most functions is increased by approximately 85%. This is due to the fact that the size of π_c is decreased accordingly. For $m = 15$, $\pi_c^{f,\uparrow^b}$ consists of 31 blocks, while π_c^f consists of 200 blocks. Notably, the calculation of score for $\pi_c^{f,\uparrow^b}$ needs 97% less gas. We achieve such a discrepancy because the process of score calculation for multiple levels demands the use of a temporary hashmap which is a storage structure. In contrast, the evaluation of the score of an individual level is performed entirely in memory.

Process		**Gas** ($\times 10^3$)	**Total**		**Gas** ($\times 10^3$)	**Total**
valid-interlinks		900	18%		120	10%
minimal-fork		1,900	39%		275	18%
args (π_s)		750	16%		750	51%
args (π_c)		950	19%		20	1%
other		400	8%		300	20%
contest	**I**	4,900	100%	**II**	1,465	100%

Table 4.5: Gas usage in contest. I: Before utilizing best-level. II: After utilizing best-level.

In Figure 4.14, we illustrate the performance gain of the client using *dispute* phase and the best-level contesting proof. The aggregated gas consumption of *submit* and *contest* phases is reduced to 3,500,000 gas. This is a critical threshold regarding applicability of the contract, since a cycle of interactions now effortlessly fits inside a single Ethereum block.

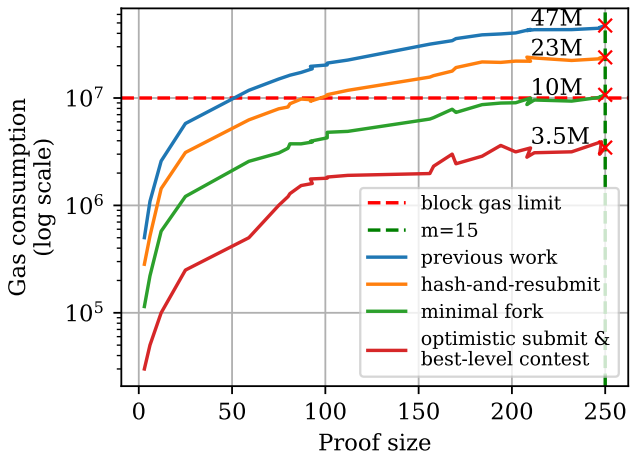


Figure 4.14: Performance improvement using optimistic schema in submit phase and best level proof in contesting proof (lower is better). Gas consumption is decreased by approximately 65%.

Algorithm 8 The NIPoPoW client enhanced with dispute phase and best-level contesting

```
1: contract crosschain
2:     ...
3:     function submit(π_s, e)
4:         require(π_s[0] = G)
5:         require(events[e] = ⊥)
6:         require(evaluate-predicate(π_s, e))
7:         events[e].hash ← H(π_s)
8:     end function
9:     function dispute(π*_s, e, i)                          ▷ i: Dispute index
10:        require(events[e] ≠ ⊥)
11:        require(events[e].hash = H(π*_s))
12:        require(¬valid-single-interlink(π_s, i))
13:        events[e] ← ⊥
14:    end function
15:    function valid-single-interlink(π, i)
16:        l ← π[i].level
17:        if π[i+1].intelink[l] = π[i] then
18:            return true
19:        end if
20:        return false
21:    end function
22:    function contest(π*_s, π_c^{f,↑^b}, e, f)
23:        require(events[e] ≠ ⊥)
24:        require(events[e].hash = H(π*_s))
25:        require(valid-interlinks(π_c^{f,↑^b}))
26:        require(minimal-fork(π*_s, π_c^{f,↑^b}, f))
27:        require(arg-at-level(π_c^{f,↑^b}) > best-arg(π*_s[f:]))
28:        require(¬evaluate-predicate(π_c^{f,↑^b}, e))
29:        events[e] ← ⊥
30:    end function
31:    function arg-at-level(π)
32:        l ← π[−1].level                                   ▷ Pick proof level from a block
33:        score ← 0                                         ▷ Set score counter to 0
34:        for b in π do
35:            if (b.level≠l) then
36:                continue
37:            end if
38:            score ← score+2^l
39:        end for
40:        return score
41:    end function
42: end contract
```

4.6 Resilience Against DOS Attacks

Chapter 5

Cryptoeconomics

We now present our economic analysis on our client. We have already discussed that the NIPoPoW protocol is performed in distinct phases. In each phase, different entities are prompted to act. As in SPV, the security assumption that is made is that at least one honest node is connected to the verifier contract and serves honest proofs. However, the process of contesting a submitted proof by an honest node does not come without expense. Such an expense is the computational power a node has to consume in order to fetch a submitted proof from the calldata and construct a contesting proof, but, most importantly, the gas that has to be paid in order to dispatch the proof to the Ethereum blockchain. Therefore, it is essential to provide incentives to honest nodes, while adversaries must be discouraged from submitting invalid proofs. In this section, we discuss the topic of incentives and treat our honest nodes as rational. We propose concrete monetary values to achieve incentive compatibility.

In NIPoPoWs, incentive compatibility is addressed by the establishment of a monetary value termed *collateral*. In the *submit* phase, the user pays this collateral in addition to the expenses of the function call, and, if the proof is contested successfully, the collateral is paid to the user that successfully invalidated the proof. If the proof is not contested, then the collateral is returned to the original issuer. This treatment incentivizes nodes to participate to the protocol, and discourages adversaries from joining. It is critical that the collateral covers all the expenses of the entity issuing the contest and in particular the gas costs of the contestation.

5.1 Collateral vs Contestation Period

The contestation period and the collateral are generally inversely proportional quantities and are both hard-coded in a particular deployment of the NIPoPoW verifier smart contract. If the contestation period is large, the collateral can be allowed to become small, as it suffices for any contester to pay a small gas price to ensure the contestation transaction is confirmed within the contestation period. On the other hand, if the contestation period is small, the collateral must be made large so as to ensure that it can cover the, potentially large, gas costs required for quick confirmation. This introduces an expected trade-off between good liveness (fast availability of cross-chain data ready for consumption) and cheap collateral (the amount of money that needs to be locked up while the claim is pending). The balance between the two is a matter of application and is determined by user policy. Any user of the NIPoPoW verifier smart contract must at a minimum ensure that the collateral and contestation period parameters are both lower-bounded in such a way that the smart contract is incentive compatible. If these bounds are not attained, the aspiring user of the NIPoPoW verifier smart contract must refuse to use it, as the contract does not provide incentive compatibility and is therefore not secure. Depending on the application, the user may wish to impose additional upper bounds on the contestation period (to ensure good liveness) or on the collateral (to ensure low cost), but these are matters of performance and not security.

5.2 Analysis

We give concrete bounds for the contestation period and collateral parameters. It is known [42] that gas prices affect the prioritization of transactions within blocks. In particular, each block mined by a rational miner will contain roughly all transactions of the mempool sorted by decreasing gas price until a certain minimum gas price is reached. We used the Etherchain explorer [16] to download recent blocks and inspected their included transactions to determine their lowest gas price. In our measurements, we make the simplifying assumption that miners are rational and therefore will necessarily include a transaction of higher gas price if they are including a transaction of lower gas price. We sampled 200 blocks of the Ethereum blockchain around March 2020 (up to block height 9,990,025) and collected their respective minimum gas prices. Starting with a range of reasonable gas prices, and based on our miner rationality assumption, we modelled the experiment of acceptance of a transaction with a given gas price within the next block as a Bernoulli trial. The probability of this distribution is given by the percentage of block samples among the 200 which have a lower minimum gas price, a simple maximum likelihood estimation of the Bernoulli parameter. This sampling of real data gives the discretized appearance in our graph. For each of these Bernoulli distributions, and the respective gas price, we deduced a Geometric distribution modelling the number of blocks that the party must wait for before their transaction becomes confirmed.

Given these various candidate gas prices (in gwei), and multiplying them by the gas cost needed to call the NIPoPoW *contest* method, we arrived at an absolute minimum collateral for each nominal gas price which is just sufficient to cover the gas cost of the contestation transaction (real collateral must include some additional compensation to ensure a rational miner is also compensated for the cost of monitoring the blockchain). For each of these collaterals, we used the previous geometric distribution to determine both the *expected* number of blocks needed to wait prior to confirmation, as well as an upper bound on the number of blocks needed for confirmation. For the purpose of an upper bound, we plot one standard deviation above the mean. This upper bound corresponds to the minimum contestation period recommended, as this bound ensures that, at the given gas price, if the number of blocks needed to wait for falls within one standard deviation of the geometric distribution mean, then the rational contester will create a transaction that will become confirmed prior to the contestation period expiring. Critical applications that require a higher assurance of success must consider larger deviations from the mean.

We plot our cryptoeconomic recommendations based on our measurements in Figure 5.1. The horizontal axis shows the collateral denominated in both Ether and USD (using ether prices of 1 ether = 246.41 USD as of June 2020). We assume that the rational contester will pay a contestation gas cost up to the collateral itself. The vertical axis shows the recommended contestation period. The solid line is computed from the block wait time needed for confirmation according to the mean of the geometric distribution at the given gas price. The shaded area depicts one standard deviation below and above the mean of the geometric distribution.

Our experiments are based on the contestation transaction gas cost of the previous section; namely they are conduced on a blockchain of 650,000 blocks with a NIPoPoW proof of 250 blocks. The contesting proof stands at a fork point after which the original proof deviates with 100 blocks, while the contesting proof deviates with 200 disjoint blocks.

We conclude that consumption of cross-chain data within the Ethereum blockchain can be obtained at very reasonable cost. If the waiting time is set to just 10 Ethereum blocks (approximately 2 minute in expectation), a collateral of just 0.50 USD is sufficient to cover for up to one standard deviation in confirmation time. Note that the collateral of an honest party is not consumed and is returned to the party upon the expiration of the contestation period. We therefore deem our implementation to be practical.

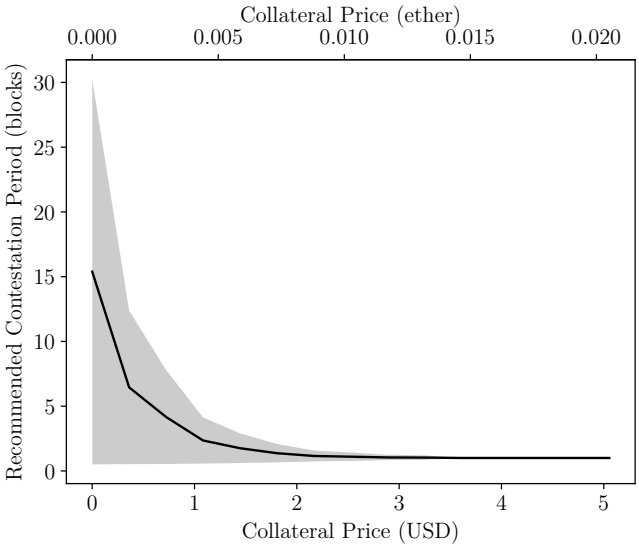


Figure 5.1: Cryptoeconomic recommendations for the NIPoPoW superlight client.

Chapter 6

Future Work

Bibliography

[1] Solidity.

[2] A. Back et al. Hashcash-a denial of service counter-measure. 2002.

[3] J. Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Assche. The keccak reference. *Submission to NIST (Round 3)*, 13:14–15, 2011.

[5] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[6] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017.

[7] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 81–84. IEEE, 2018.

[8] A. Chepurnoy. Aurora coin, 2014.

[9] A. Chepurnoy. Ergo platform, 2017.

[10] A. Chepurnoy, C. Papamanthou, and Y. Zhang. Edrax: A cryptocurrency with stateless transaction validation. *IACR Cryptology ePrint Archive*, 2018:968, 2018.

[11] J. Chow. BTC Relay, Dec 2014.

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

[13] ConsenSys. A Guide to Events and Logs in Ethereum Smart Contracts, June 2016.

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

[15] W. Entriken. Eip2028, 2019.

[16] Etherchain. Etherchain, Jun 2020.

[17] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[18] K. Floersch. Ethereum smart contracts in l2: Optimistic rollup, August 2019.

[19] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310, 2015.

[20] A. Gluchowski. Optimistic vs. zk rollup: Deep dive, November 2019.

[21] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[22] A. H. Off-Chain Data Storage: Ethereum & IPFS, October 2017.

[23] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[24] M. Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[25] K. Karantias, A. Kiayias, and D. Zindros. Smart contract derivatives. In *The 2nd International Conference on Mathematical Research for Blockchain Economy*. Springer Nature, 2020.

[26] A. Kiayias, N. Lamprou, and A.-P. Stouka. Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 61–78. Springer, 2016.

[27] A. Kiayias, A. Miller, and D. Zindros. Non-Interactive Proofs of Proof-of-Work. In *International Conference on Financial Cryptography and Data Security*. Springer, 2020.

[28] A. Kiayias and D. Zindros. Proof-of-work sidechains. In *International Conference on Financial Cryptography and Data Security*. Springer, Springer, 2019.

[29] B. Laurie, A. Langley, and E. Kasper. Rfc6962: Certificate transparency. *Request for Comments. IETF*, 2013.

[30] Y. Lu, Q. Tang, and G. Wang. Generic superlight client for permissionless blockchains. *arXiv preprint arXiv:2003.06552*, 2020.

[31] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 16–30, 2019.

[32] R. 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.

[33] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[34] T. Nolan. Alt chains and atomic transfers, May 2013.

[35] A. Polydouri, A. Kiayias, and D. Zindros. The velvet path to superlight blockchain clients, 2020.

[36] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[37] Tak. Store data by logging to reduce gas cost, October 2019.

[38] N. Team. Nimiq, 2018.

[39] W. Team. Webdollar - currency of the internet, 2017.

[40] P. Todd. Merkle mountain ranges, October 2012.

[41] F. Volland. Memory Array Building, April 2018.

[42] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[43] K. Wüst and A. Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

[44] A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, and W. 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.

[45] D. Zindros. *Decentralized Blockchain Interoperability*. PhD thesis, Apr 2020.