A gas-efficient superlight Bitcoin client in Solidity

May 1, 2020

Abstract

Superlight clients enable the verification of proof-of-work-based blockchains by checking only a small representative number of block headers instead of all the block headers as done in SPV. Such clients can be embedded within other blockchains by implementing them as smart contracts, allowing for cross-chain verification. One such interesting instance is the consumption of Bitcoin data within Ethereum by implementing a Bitcoin superlight client in Solidity. While such theoretical constructions have demonstrated security and efficiency, no practical implementation exists. In this work, we put forth the first practical Solidity implementation of a superlight client which implements the NIPoPoW superblocks protocol. Our implementation is open source and we demonstrate it is production-ready by providing full test coverage. Contrary to previous work, our Solidity smart contract achieves sufficient gas-efficiency to allow a proof and counter-proof to fit within the gas limit of a block, making it practical. We provide extensive experimental measurements for gas. The optimizations that enable gas-efficiency heavily leverage a novel technique which we term hash-and-resubmit, which almost completely eliminates persistent storage requirements, the most expensive operation of smart contracts in terms of gas. Instead, the contract asks contesters to resubmit data and checks their veracity by hashing it. We show that such techniques can be used to bring down gas costs significantly and may have applications to other contracts. We also identify and rectify multiple implementation security issues of previous work such as premining vulnerabilities. Lastly, our implementation allows us to calculate concrete cryptoeconomic parameters for the superblocks NIPoPoWs protocol and in particular to make recommendations about the monetary value of the collateral parameters. We provide such parameter recommendations over a variety of liveness and adversarial bound settings.

Contents

Chapter 1

Introduction

1.1 Motivation

Bitcoin 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 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, 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.

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. After Bitcoin and Ethereum, the most important blockchains in terms of capitalization are Ripple, Tether, Bitcoin Cash and Litecoin. 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

To perform crosschain operations between two chains, there must be some mechanism to communicate to chain A that an event occurred to chain B,

[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.

such that a payment tool place. One trivial way for Alice to determine if an event took place in chain A and register it to chain B is to participate in both chains. But this way is very inefficient because the storage needed to store the blockchain is sizable, and it grows with time. At the moment, Bitcoin is 242.39 GB and Ethereum has exceeded 1 TB. Naturally, it is impractical for every user to accommodate this size of data.

An early solution to the extensive space of blockchain was given by Satoshi, and is called Simplified Payment Verification (SPV) protocol. In SPV, only the headers of blocks are stored, saving of a lot of storage. However, even with this protocol, Bitcoin headers have the total size of 50 GB (80 bytes each). A mobile client needs several minutes, even hours to download all information needed to function as an SPV client. Moreover, not all blockchains have small-sized headers like Bitcoin. Block headers of Ethereum, for instance, are 508 Bytes and for different altcoins they span several MB of data. SPV clients lead to unpleasant user experience at some cases, while they are completely impractical in others.

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.

1.3 Related Work

Non-Interactive Proofs of Proof of Work (NIPoPoWs)(ref) is the fundamental building block of our solution. This cryptographic primitive is provably secure and provide succinct proofs regarding the occurrence of an event in a chain. The size of the proofs is logarithmic to the size of the underlying blockchain, which means that they are growing slowly compared to the blockchain growth rate.

Christoglou (ref), has provided a Solidity smart contract which is the first ever implementation of crosschain events verification based on NIPoPoWs, where proofs are submitted and checked their validity. This solution, however, is impossible to apply to the real blockchain due to extensive usage of resources and important security vulnerabilities.

A different aspect to solve interoperability is a protocol introduced by Summa team. In this work users waits for a transaction to be buried under several blocks which implies that the transaction must belong to the real

chain, and thus be valid. This treatment enables fast and cheap crosschain capabilities. But this solution has not been proved cryptographically secure. In fact, an attack has been laid out that makes the protocol vulnerable to non-rational adversaries.

1.4 Our contributions

Add later on: A series of keen observations, the application of gas-efficient practices and the utilization of modern Solidity features led us to the design of a new verifier architecture.

We put forth the following contributions:

(a) We developed the first ever decentralized client that securely verifies crosschain events and is applicable to the real blockchain. Our client establishes a safe, cheap and trust-less solution to the interoperability problem. Our client is implemented in Solidity and verifies Bitcoin events to the Ethereum blockchain.

(b) We prove via application that NIPoPoWs can be utilized in the real blockchain, making the cryptographic primitive the first ever applied construction of succinct proofs to a real setting.

Our implementation meets the following requirements:

(a) Security: The client is invulnerable against all adversarial attacks.

(b) Is trust-less: The client is not dependent on third-party applications and operates in a fully transparent, decentralized manner.

(c) Applicability: The client can be utilized in the real blockchain and comply with all environmental constraints, i.e. block gas limit and calldata size of Ethereum blockchain.

(d) Is cheap: The application is cheaper to use than the current state of the art technologies. We still need to show this

We selected Bitcoin as source blockchain as it the most used cryptocurrency and enabling crosschain transactions in Bitcoin is beneficial to the vast majority of blockchain community. We selected Ethereum as the target blockchain because it is also very popular to the community and it supports smart contracts which is a requirement in order to perform on-chain verification.

Some applications that demonstrate the usage of our client are:

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

1.5 Structure

In Section 2 we description of all relevant background technologies and previous work. In Section 3 we put forth the implementation of our client. In Section 4 we show our cryptoeconomic analysis and, finally, in Section 5, we discuss applications of our client and future work.

Chapter 2

Background

Relevant technologies

2.1 Primitives

Describe primitives

2.2 Bitcoin

Describe Bitcoin blockchain

2.3 Ethereum

Describe Ethereum blockchain

2.3.1 Solidity

Describe the use of solidity language

2.3.2 Smart contracts

Describe the use of smart contracts

2.3.3 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a sandboxed virtual stack embedded within each full Ethereum node, responsible for executing contract bytecode. Contracts are typically written in higher level languages, like Solidity, then compiled to EVM bytecode.

This means that the machine code is completely isolated from the network, filesystem or any processes of the host computer. Every node in the

Ethereum network runs an EVM instance which allows them to agree on executing the same instructions. The EVM is Turing complete, which refers to a system capable of performing any logical step of a computational function. JavaScript, the programming language which powers the worldwide web, widely uses Turing completeness.

Ethereum Virtual Machines have been successfully implemented in various programming languages including C++, Java, JavaScript, Python, Ruby, and many others.

The EVM is essential to the Ethereum Protocol and is instrumental to the consensus engine of the Ethereum system. It allows anyone to execute code in a trustless ecosystem in which the outcome of an execution can be guaranteed and is fully deterministic (i.e.) executing smart contracts.

2.4 Environment Set-Up

2.4.1 Existing Environments

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(ref), Remix(ref) and Embark(ref). 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(ref) 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(ref).

2.4.2 Solidity Compiler

2.4.3 Web3

The components we used as building blocks are Web3(ref) which is a powerful library for interacting with Ethereum, the Solidity v0.6.6 compiler(ref), and EthereumTester which is a set of tools for testing Ethereum-based applications. For the purpose of our project, a private blockchain running an Ethereum Virtual Machine (EVM) was deployed. This is a common practice for Ethereum development since it greatly facilitates testing procedures. Our environment supports multiple EVMs, namely Geth(ref), Ganache(ref) and Py-EVM(ref).

2.4.4 Ethereum Virtual Machines

All aforementioned EVMs deliver implementations that comply with the specifications described at the Ethereum yellow paper(ref). However, different implementations provide unique choices to the developer, each of which helped us to progress effortlessly during different stages of our work.

Py-EVM:

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(ref).

Ganache:

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:

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.

Configuration:

We observed 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(ref). We hope this will facilitate future work.

figure of Py-EVM vs Ganache vs Geth

2.4.5 Gas Profiling

extend this. Show other existing profiling tools, why we used this Yushi-sama

Another useful utility we used is solidity-gas-profiler(ref), a profiling utility by Yushih. 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.

figure of gas profiling

2.5 Non-Interactive Proofs Of Proof Of Work

Describe the rationale behind NIPoPoWs, what they provide

2.5.1 Prefix Proofs

Describe prefix proofs

2.5.2 Suffix Proofs

Describe suffix proofs

2.5.3 Infix Proofs

Describe infix proofs

2.6 Forks

Soft, hard and velvet fork

2.7 Previous Work

2.7.1 Methodology

As mentioned above, we used a previous verifier implementation(ref) as a basis for our implementation. Since we adopted common primitives, we used some of the tools Giorgos et al. used for functionalities such as constructing blockchains and proofs. For the purposes of our project, we needed 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 that our work shares this the previous implementation. This includes the following::

1. Construct a blockchain
2. Construct a proof for an event in the blockchain
3. Verify the proof

Blockchain

The tool that creates the blockchain was created by Andrew Miller, one of the writers of Non-Interactive Proofs of Proof of Work(ref) paper. The tool is using the Bitcoin library(ref) to construct a blockchain similar to Bitcoin's. The interlink pointers are organized into a Merkle(ref) tree and the index is determined by their level. For details regarding the level calculation, see section(ref). 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.

figure of blockchain

Superblock Levels

We assume that the difficulty target of mined blocks is constant. As discussed in section(ref), 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.

do we need to justify this?

Proof

The tool that creates proofs was also created by Andrew Miller. The prover receives the following inputs:

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

Security parameters k, m are part of the NIPoPoW model and are explained in section(ref).

The prover's output is a Proof of Proof of Work that satisfies the above security parameters. The prover needed to be enhanced in order to create special test cases (section(ref)) and enable our optimized architecture (section(ref)).

figure of proof

Verifier

The goal of the verifier is to securely determine if an event has occurred in the honest blockchain. For this, the concept of NIPoPoWs is used. A proof is submitted in combination with a predicate. The proof is considered valid if it is constructively correct(ref) and the predicate is true for the chain described by the proof. The predicate represents the existence of an event in the source blockchain, such as the occurrence of a transaction. In our case, the predicate indicates the existence of a block in the proof.

How much more demanding is to prove an actual transaction?

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 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 entity successfully contests the submitted proof at some round $r_c < r$.

Contesting phase:

In **contesting phase**, a new proof is submitted. If this proof is better, 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 expected 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 should be unsuccessful and the *predicate* should remain *true*.

- If an *adversary* submits a proof, then an *honest* party should make a contest. The contest should invalidate the original submission and the *predicate* should become $false$.
- The scenario in which an *adversary* submits a proof an *honest* does not contest should not take place due to the assumption that at least one honest party observes the traffic of the contract.

2.7.2 Notation

In this section we will use the notation displayed in table **??**.

2.7.3 Implementation

Should it be Previous Work?

In this subsection we present previous work. We prepare the reader by focusing at specific aspects in which our solution differs. We will later show these differences and we will analyse on how they impact the results of our application.

Overview

As mentioned in the NIPoPoWs(ref Algorithm 7) paper, in order to construct a verifier, a Directed Acyclic Graph (DAG) needs to be maintained in memory. This structure is stored in the form of a hashmap(ref), 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 **??**. The DAG is then used to construct ancestors structure by performing a simple graph search. By iterating ancestors, we can securely determine the value of the predicate.

This logic is intuitive and efficient to implement in most traditional programming languages (C++, JAVA, Python, JavaScript, etc). However, such an algorithm cannot be efficiently implemented in Solidity as is. This is not due to the lack of features, such as the existence of hashmaps, but because Solidity treats storage differently than most programming languages. As mentioned above(ref) in smart contracts the caller needs to pay in gas for the execution of operations such as accessing and storing data. Reading from and writing to persistent memory are very expensive operations in Solidity, as stated in the Ethereum yellow paper(ref). A summary of gas costs for storage and memory access if is displayed in Table **??**. This fact was observed by Giorgos et al. and was recognized as the bottleneck of the application.

Solidity Algorithms

We describe each phase of the previous implementation in Algorithms **??** and **??**. We highlight structures that access persistent memory. Note that deleting from persistent memory is also considered a storage operation

2.8 Difficulty

Describe constant and non-constant difficulty

Chapter 3

Implementation

In this chapter, we put forth out implementation. First, we present our analysis of the previous work. Then we describe our

3.1 Analysis of Previous Work

In this section, we analyse the verifier by 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. Finally, we present the vulnerabilities we discovered, and how we mitigated them in our work.

3.1.1 Porting from old Solidity version

We used the latest version of Solidity compiler for our analysis. To perform the 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`, explicit `memory` and `storage` declaration and explicit cast from `address` to `payable address`. We also used our configured EVMs with EIP 2028 enabled to benefit from low cost function calls. The functionality of the contract remained equivalent.

3.1.2 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.

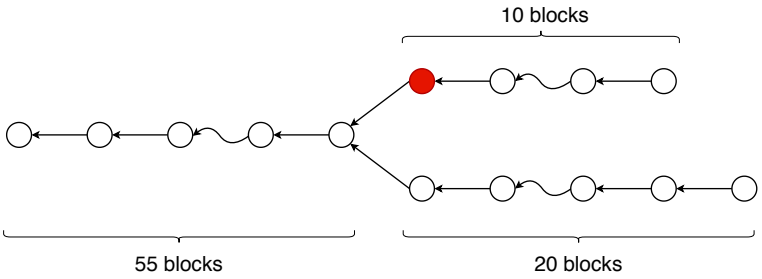


Figure 3.1: The red block indicates the block of interest. Curved connections imply intermediate blocks. The adversary creates a proof for an event that does not exist in the honest chain

Submit function	**gas usage**
validate Interlink	158597
store proof	522,885
store DAG	1,520,992
store ancestors	2,408,025
evaluate predicate	322,429
delete ancestors	169,874
Sum	10,002,974

Contest function	**gas usage**
validate Interlink	213,810
find LCA	797,358
compare proofs	280,758
store proof	293,445
update DAG	1,266,348
store ancestors	3,176,599
evaluate predicate	343,056
delete ancestors	136,580
Sum	12,922,774

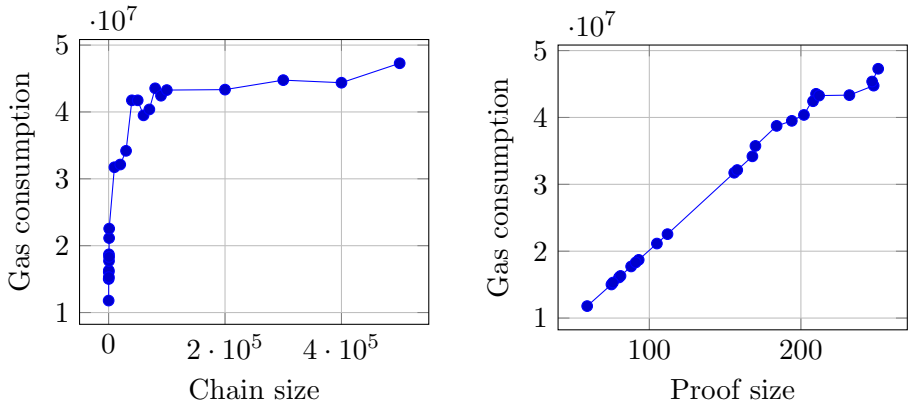
Table 3.1: Execution for proof of 25 blocks

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 selected 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 500,0000-blocks chain demands 47,280,453 gas. In Figure 3.2, we show gas consumption for submit phase for different chain sizes and their corresponding proofs sizes. We demonstrate results for chains sizes from 100 blocks (corresponding proof size 25) to 500,000 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 **??** insert figure for proof size with relation to the chain in Back-

ground, 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 an average 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 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

3.1.3 Security Analysis

Pre-mining

We observed that the verifier is vulnerable to pre-mining(ref). By definition, a NIPoPoW is structurally correct if two properties are satisfied:

(a) The interlink structure of all blocks is correct. This is to prevent adversaries from injecting blocks that do not exist in the original blockchain.

(b) The first block of proof is *genesis*. This is to prevent adversaries from create coins before blockchain are advertised at the public network.

The second property is not verified in the previous work, exposing the verifier to pre-mining attacks. We can easily mitigate this vulnerability by initializing the smart contract with the *genesis* block of the blockchain we will use and add an assertion in submit and contest phase that proofs need to satisfy property (b). The needed changes are shown in Algorithms 1 and 2.

Algorithm 1: Contract Constructor

Input: *genesis* block

1 $genesis_s \leftarrow genesis$

...

Algorithm 2: Submit Event Proof

Input: $proof$, ...

1 require $proof[0] = genesis_s$

...

Score Calculation

During our tests, we observed that the calculation of proofs score was incorrect. The score of each level is needed to determine which proof represents the chain with the most Proof of Work. Between two proofs, we only need to calculate the score starting from their *lca* until the tip of each proof. Different levels are needed because the *lca* between two proofs is only known when the contesting proof is submitted. The security parameter m needs to be satisfied for every sub-proof $\pi[:lca]$. We ensure that this is $true$ by creating proofs of multiple levels, so that security parameter m applies, disregarding *lca*'s position.

Figure for the need of multiple levels

Each block has a level, calculated as describe in Section(ref)

$$level = getLevel(block)$$

Consequently, each level of the proof consists of a number of blocks n_{level}. This number is the sum of blocks of level $\geq level$, i.e. block of level l are also blocks of levels $l-1$, $l-2$, etc. The score of each level is computed as:

$$score_{level} = 2^{level} \times n_{level}$$

After running out tests for the previous implementation, we observed that function $getLevel(block)$ of the contract was returning $block.level - 1$ instead of $block.level$ resulting to incorrect score computation. This can prevent an honest party from successfully contesting an adversarial proof, making the contract insecure. The function was refined to return the correct value.

3.2 Storage Elimination

Should this be a new section?

As mentioned above, the bottleneck we had to eliminate was the extensive usage of storage. We created a new architecture that allow us to discard all expensive store operations and utilize memory instead. This led to massive decrease of gas consumption. In this section, we present the difference in gas usage between storage and memory utilization, and how a NIPoPoW verifier can be implemented in Solidity without persisting proofs.

3.2.1 Storage vs Memory

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

```
1  pragma solidity ^0.6.6;
2
3  contract StorageVsMemory {
4      uint256 size;
5      uint256[] storageArr;
6
7      constructor(uint256 _size) public {
8          size = _size;
9      }
10
11     function withStorage() public {
12         for (uint i = 0; i < size; i++) {
13             storageArr.push(i);
14         }
15     }
16
17     function withMemory() public view {
18         uint256[] memory memoryArr = new uint256[](size);
19         for (uint256 i = 0; i < size; i++) {
20             memoryArray[i] = i;
21         }
22     }
```

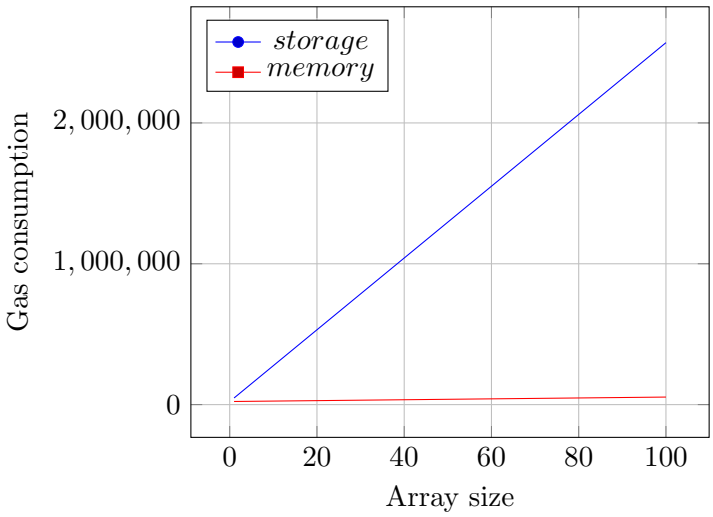


Figure 3.3: Gas consumption for memory and storage

```
23 }
```

Listing 3.1: Solidity test for storage and memory

Highlight code

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 3.3. For `size` = 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 Constantinople(ref) 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 users.

3.2.2 Making use of calldata

In previous work we needed to store submitted proofs in order to proceed to contest. In this subsection we show an approach to securely verify proofs without utilizing the persistent storage of the smart contract.

The rationale is to demand from the caller to provide two proofs to the contract during contest phase: (a) π_{exist}, which is a copy of the originally

[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.

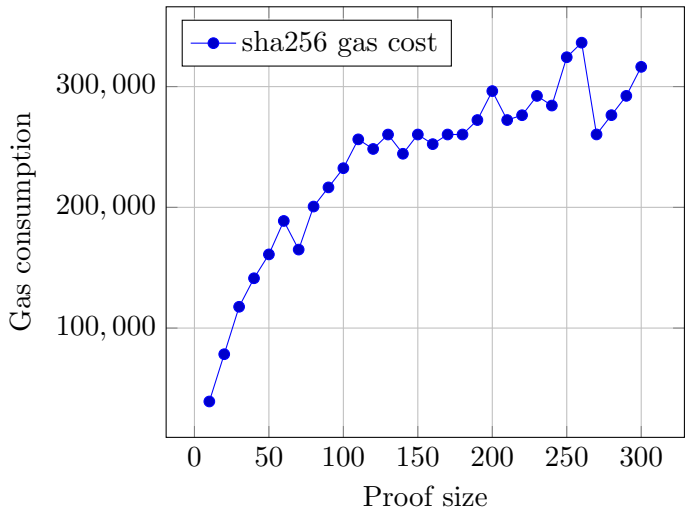


Figure 3.4: Gas consumption for hashing proofs and storing digest

submitted proof π_{orig}, and (b) π_{cont}, which is the contesting proof. Proof π_{orig} can be retrieved by observing contract's *calldata*. We prevent an adversary from malforming π_{exist} by storing the hash of π_{orig} to contract's state during submit phase and then verifying that π_{exist} has the same hash. The operation of hashing the proof and storing the digest is cheap[2] as shown in figure 3.4. We calculate the digest of the proof by:

```
digest = sha256(abi.encodePacked(proof))
```

The size of the digest of a hash is 32 bytes. To persist such a small value in contract's memory only adds a constant, negligible cost overhead to our implementation.

3.2.3 Removing DAG and ancestors

As shown in table 3.1, the most demanding operation is the creation and population of DAG and ancestors. In this subsection we show how these two structures can be discarded from the verifier.

Using subset

Our first realization was that instead of storing the DAG of π_{exist}, π_{cont}, we can require

$$\pi_{exist}\{: lca_e\} \subseteq \pi_{cont}\{: lca_c\}$$

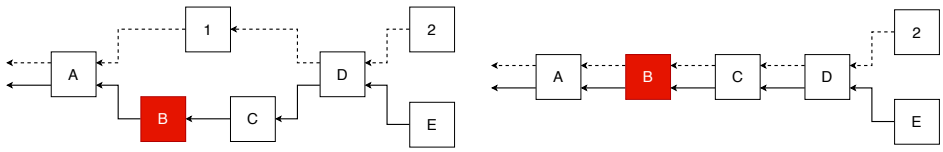
[2]By setting $k = 6$, $m = 13$, a proof for the entire Bitcoin blockchain consists of less than 300 superblocks. The hashing of such a proof costs approximately 300,000 gas units.

where lca_e and lca_c are the indices of the lca block in π_{exist}, and π_{cont}, respectively. This way we avoid the demanding need of composing auxiliary structures DAG and ancestors on-chain. The implementation of `subset` is displayed in listing 3.2. The complexity of the function is

$$\mathcal{O}(\mid \pi_{exist}[: lca_e] \mid + \mid \pi_{cont}[: lca_c] \mid)$$

```
1  function subset(
2      Proof memory exist, uint existLca,
3      Proof memory cont, uint contLca
4  ) internal pure returns(bool)
5  {
6      uint256 j = contLca;
7      for (uint256 i = existLca; i < exist.length; i++) {
8          while (exist[i] != cont[j]) {
9              if (++j >= contLca) { return false; }
10         }
11     }
12     return true;
13 }
```

Listing 3.2: Implementation of subset



(a) Valid π_{cont} before using subset

(b) Valid π_{cont} after using subset

Figure 3.5: Blocks connected with solid lines indicate π_{exist} and blocks connected with dashed lines indicate π_{cont}

The gas consumption difference between *subset* and $DAG + ancestors$ is displayed at figure 3.6. *Subset* solution is approximately 2.7 times more efficient.

Subset complexity and limitations

Requiring π_{exist} to be a subset of π_{cont} greatly reduces gas, but the complexity of the *subset* algorithm is high since both proofs have to be iterated from genesis to their respective lca index. Generally, we expect for 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 is exponentially weakened as the honest chain progresses. This means that the length of π, $\mid \pi \mid$ is be considerably close to $\mid \pi[: lca] \mid$, and the complexity of `subset()` is effectively $\mathcal{O}(2 \mid \pi \mid)$.

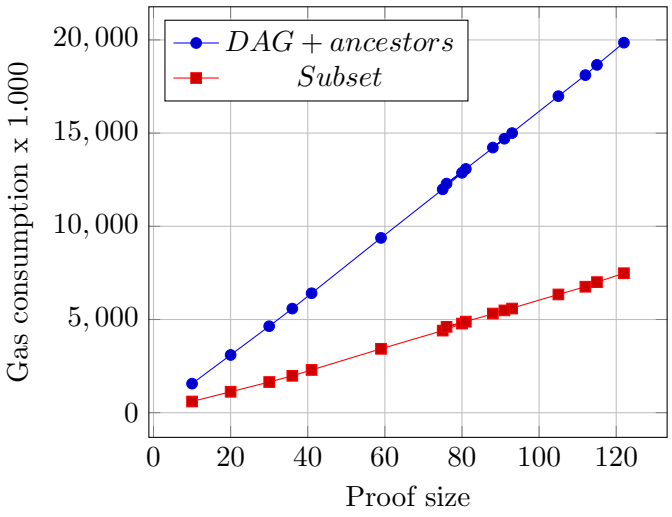


Figure 3.6: Gas consumption for DAG+ancestors and subset

In realistic cases, where the *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 blockchains since it exceeds the block gas limit of the Ethereum blockchain by far.

Position of block of interest

By analyzing the benefits and trade-offs of *subset*, we concluded that there is a more efficient way to treat storage elimination. In general, the concept of *subset* facilitated the case in which the block of interest belongs in the sub-proof $\pi_{exist}[: lca_e]$. But in this case, both π_{exist} and π_{cont} contain the block of interest at some index, as can be seen in figure 3.5b. Consequently, π_{cont} cannot contradict the existence of the block of interest and the predicate is evaluated *true* for both proofs. This means that if (a) π_{exist} is structurally correct and (b) the block of interest is in $\pi_{exist}[: lca_e]$, then we can safely conclude that contesting with π_{cont} is redundant. Therefore, $E_{contest}$ can simply send π_{cont}[lca:] to the verifier. The truncation of π_{cont} to $\pi_{cont}[lca_c :]$ can be easily addressed from E_{cont}, since π_{exist} is accessible from the contract's calldata and both proofs can be iterated off-chain.

Disjoint proofs

We will refer to the truncated contesting proof as π^{tr}_{cont} and to lca_e simply as *lca*. For the aforementioned, the following statements are true:

(a) $\pi_{exist}[0] = genesis$

(b) $\pi_{exist}[lca] = \pi^{tr}_{cont}[0]$

The requirement that needs to be satisfied is

$$\pi_{exist}\{lca + 1 :\} \cap \pi^{tr}_{cont}\{1 :\} = \emptyset$$

The implementation of this operation is shown in listing 3.3.

```
function disjoint(
    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 3.3: Implementation for disjoint proofs

The complexity of `disjoint()` is

$$\mathcal{O}(\mid \pi_{exist}[lca_e :] \mid \times \mid \pi^{tr}_{cont} \mid)$$

3.3 Fixing vulnerabilities and restricting gas usage

Chapter 4

Results

Chapter 5

Conclusion

Chapter 6

Future Work