# A gas-efficient superlight Bitcoin client in Solidity

April 22, 2020

**Abstract**

Place abstract here

## 1 Introduction

### 1.1 Motivation

Digital coins are peer-to-peer currencies based on applied cryptography for the validation of transactions. Most of them are based on blockchain(ref), a form of decentralized database. In this database, a public ledger is deployed which is stored and updated by thousands of users in absence of supervision from public authorities.

In 2008, Bitcoin(ref), the first ever successful decentralized digital coin, was invented by an unknown person or group of people using the name Satoshi Nakamoto. A year after, the bitcoin network started, quickly followed by several other digital coins, which in the cryptocurrency folklore are known as altcoins. Usually, altcoins are based on innovative features previously missing from the cryptocurrency market, and they are either accepted or rejected by the community. Popular altcoins are Ethereum(ref), which is the first to provide smart contracts, Ripple(ref) which provides real-time payment settlements and Litecoin(ref) which enables near-zero cost payments.

Over the last decade, cryptocurrencies gained attention from the public as an increased number of users accept and trust decentralized transactions. Specifically, in 2017, the popularity of cryptocurrencies rapidly grew, resulting in massive capitalisation and creation of tokens. During this period, some of the issues that blockchain technology faces were displayed. One of these issues is blockchain interoperability, the property of distinct blockchains to interact efficiently with each other. Despite its great importance, this field has not been addressed until recently. To date, cryptocurrencies are lacking a commonly accepted protocol that enables distributed interoperability. Such a protocol would be very useful to blockchain technology, since it would allow users to variously utilize features of different blockchains. For example, one can store their funds in Bitcoins, and convert them to Ether to make

a payment, benefiting from lower transaction fees and quicker transaction rates.

A crosschain protocol would enable two main operations

- Crosschain trading: An entity with deposits in blockchain A, makes a payment to an entity at blockchain B.
- Crosschain fund transfer: Entity transfers owning funds from blockchain A to blockchain B. After this operation, the funds no longer exist at blockchain A. The entity can return any portion of the original amount to the blockchain of origin.

Currently, this operation is only available to the users via third party applications, such as multi-currency wallets. This treatment certainly opposes to the nature of blockchain, which is a decentralized construction. This motivated us to create a solution that enables cheap, trust-less crosschain operations.

### 1.2 Previous Work

In this paper, we focus on recent research in the area of crosschains. In particular, we make use of the cryptographic primitive Non-Interactive Proofs of Proof of Work (NIPoPoWs)(ref), which enables the compression of a chain to its poly-logarothmic size. NIPoPoWs is the main building block of our solution in order to make the occurrence of an event of blockchain A provably known to blockchain B.

We are based on previous work done by Giorgos Christoglou et al.(ref), which was the first ever implementation of crosschain events verification. The work of Giorgos et al. focuses on verifying Bitcoin events from the Ethereum blockchain. In order to provide this functionality, a NIPoPoW verifier was developed in Solidity(ref), one of the programming languages of Ethereum blockchain. This solution, however, is impossible to be applied in a real blockchain due to extensive gas usage and severe security issues.

maybe "severe" is too harsh

### 1.3 Our contributions

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. This allowed us to repair previous vulnerabilities and enable crosschain operations by providing a secure, superlight Bitcoin client that can be deployed on the real blockchain.

Our contribution is the creation of a Bitcoin client in Solidity that verifies events across blockchains while meeting the following criteria:

- Security: Is secure against any adversarial attack.

- Trustless: Does not have dependencies at any third-party applications.
- Applicability: Is applicable to the real blockchain. That is, data derived from the full-sized Bitcoin blockchain are successfully accepted and processed by the client without exceeding the build-in constraints of the Ethereum blockchain (i.e. block gas limit, calldata limit etc).
- Cheap: Is cheaper than the current state of the art technologies. This would make trustless crosschain transactions more popular and affordable.

# 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 Smark 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 Non-Interactive Proofs Of Proof Of Work

Describe the rationale behind NIPoPoWs, what they provide

### 2.4.1 Prefix Proofs

Describe prefix proofs

### 2.4.2 Suffix Proofs

Describe suffix proofs

### 2.4.3 Infix Proofs

Describe infix proofs

## 2.5 Forks

Soft, hard and velvet fork

## 2.6 Difficulty

Describe constant and non-constant difficulty

# 3 Implementation

## 3.1 Environment Set-Up

### 3.1.1 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).

### 3.1.2 Dependencies

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).

### 3.1.3 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 inly 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 using an EVM alone 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

#### 3.1.4 Gas Profiling

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

### 3.2 Model

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

### 3.2.1 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

### 3.2.2 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?

### 3.2.3 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

### 3.2.4 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.

### 3.2.5 Notation

In this section we will use the notation displayed in table 1.

| Symbol | Description |
|---|---|
| $E_{submit}$ | The entity submitting a proof at submit-phase |
| $E_{cont}$ | The entity initiating a proof contest at contest-phase |
| $\pi_{orig}$ | The proof submitted by S |
| $\pi_{exist}$ | The copy of $\pi_{orig}$ submitted by C |
| $\pi_{cont}$ | The contesting proof submitted by C |
| $\mid \pi \mid$ | The size of $\pi$ |
| $\pi$[i] | The $i$-th superblock of $\pi$ |
| $\pi$[i:j] | The aggregation of superblocks $\pi[i]$, $\pi[i+1]$, ..., $\pi[j-1]$ |
| $\pi$[i:] | The aggregation of superblocks from $\pi[i]$ until the last block of $\pi$ |
| $\pi$[:j] | The aggregation of superblocks from the first block of $\pi$ until the $\pi[j-1]$ |
| $\pi\{i, j\}$ | The set of superblocks consisting of $\pi[i, j]$ |

Table 1: Notation

## 3.3 Previous 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.

In the process of building our Bitcoin Client, a suit of thorough unit tests were written to assert the correction of our results which we also used at the previous work. We discovered some erroneous functionalities that we appose in subsection **??** tests should be more visible

### 3.3.1 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.

| 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: Ethereum gas list

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 2. This fact was observed by Giorgos et al. and was recognized as the bottleneck of the application.

### 3.3.2 Phases

We describe each phase of the previous implementation in Algorithms 1 and 2. We highlight structures that access persistent memory. Note that deleting from persistent memory is also considered a storage operation

**Algorithm 1:** Submit Event Proof

**Input:** $proof$, $predicate$
**Data:** `array` $proof_s$, `hashmap` $DAG_s$, `bool` $predicate_s$

1 require $predicate_s = \emptyset$
2 require $validInterlink(proof)$
3 $DAG_s \leftarrow DAG_s \cup proof$
4 $proof_s \leftarrow proof$
5 $ancestors_s \leftarrow findAncestors(DAG_s)$
6 $predicate_s \leftarrow evaluatePredicate(ancestors_s, predicate)$
7 delete $ancestors_s$

**Algorithm 2:** Submit Contesting Proof

**Input:** $proof'$, $predicate$
**Data:** `array` $proof_s$, `hashmap` $DAG_s$, `bool` $predicate_s$

require $predicate_s = predicate$
$lca \leftarrow$ findLca($proof_s$, $proof'$)
require $score(proof'[lca:]) > score(proof_s[lca:])$
$DAG_s \leftarrow DAG_s \cup proof'$
$ancestors_s \leftarrow findAncestors(DAG_s)$
$predicate_s \leftarrow evaluatePredicate(ancestors_s, predicate)$
delete $ancestors_s$

### 3.3.3 Gas analysis

Here, we layout experiments that show the gas usage of the previous implementation. In these experiments, we used a small proofs[1]. These sizes are unrealistic for a real blockchain, but are helpful to demonstrate the gas of the functions of the contract for each phase. The gas expend for submit and contest for proof of 20 blocks is shown in Table **??**. We observe that even for small proofs, the gas cost is high - 4 million gas units, given that the block gas limit of Ethereum is currently at 9.908.813(ref) gas units. This is mainly due to the extensive storage usage.

maybe show numbers for the minimum proof that exceeds block gas limit.

### 3.3.4 Security Analysis

**Pre-mining** We observed that the smart contract is vulnerable to pre-mining(ref). By definition, a valid 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 3 and 4.

[1]For sizes of proofs for realistic blockchain sizes, refer to Section(ref)

**Algorithm 3:** Contract Constructor

**Input:** *genesis* block

1 $genesis_s \leftarrow genesis$

...

**Algorithm 4:** 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.

### 3.4 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.

### 3.4.1 Storage vs Memory

We will first demonstrate the difference in gas usage between storage and memory for a smart contract in Solidity. Suppose 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 (uint 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 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 1. 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[2]. 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.

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

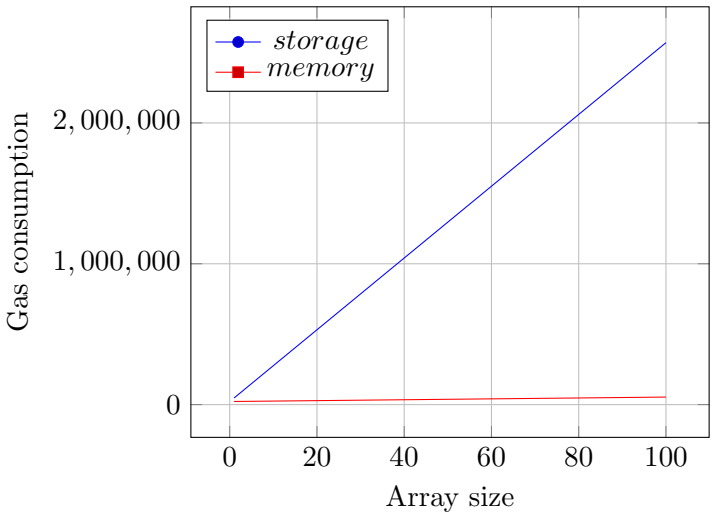



Figure 1: Gas consumption for memory and storage

### 3.4.2 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) $\pi_{exist}$, which is a copy of the originally submitted proof $\pi_{orig}$, and (b) $\pi_{cont}$, which is the contesting proof. Proof $\pi_{orig}$ can be retrieved by observing contract's *calldata*. We prevent an adversary from malforming $\pi_{exist}$ by storing the hash of $\pi_{orig}$ to contract's state during submit phase and then verifying that $\pi_{exist}$ has the same hash. The operation of hashing the proof and storing the digest is cheap[3] as shown in figure 2. 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.

### 3.4.3 Removing DAG and ancestors

As shown in table **??**, 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.

[3]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.

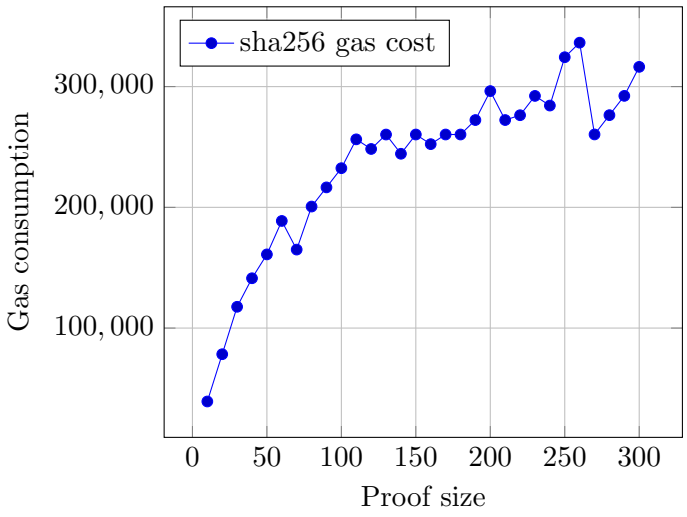



Figure 2: Gas consumption for hashing proofs and storing digest

**Using subset** Our first realization was that instead of storing the DAG of $\pi_{exist}$, $\pi_{cont}$, we can require

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

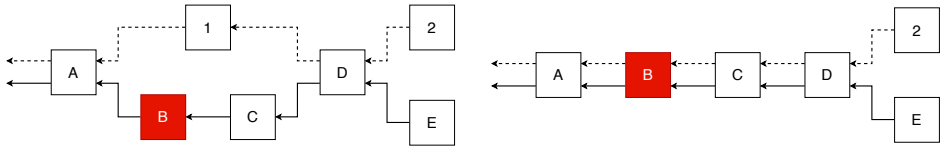
where $lca_e$ and $lca_c$ are the indices of the $lca$ block in $\pi_{exist}$, and $\pi_{cont}$, respectively. This way we avoid the demanding need of composing auxiliary structures DAG and ancestors on-chain. The implementation of `subset` is displayed in listing 2. The complexity of the function is

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

```
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 2: Implementation of subset

The gas consumption difference between *subset* and $DAG + ancestors$ is displayed at figure 4. *Subset* solution is approximately 2.7 times more efficient.




(a) Valid $\pi_{cont}$ before using subset (b) Valid $\pi_{cont}$ after using subset

Figure 3: Blocks connected with solid lines indicate $\pi_{exist}$ and blocks connected with dashed lines indicate $\pi_{cont}$

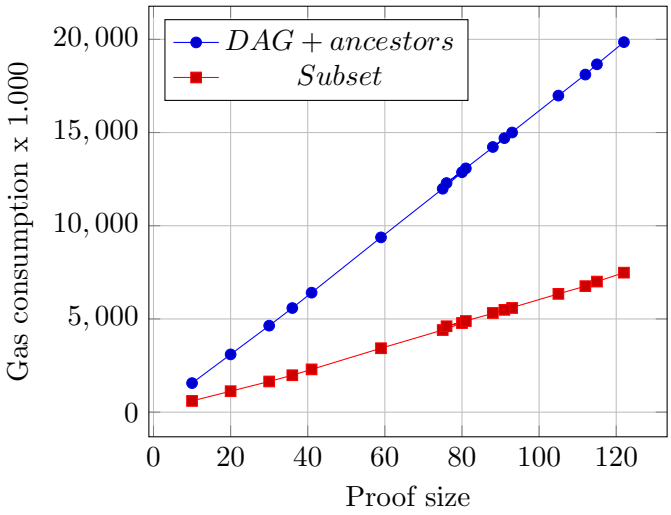



Figure 4: Gas consumption for DAG+ancestors and subset

**Subset complexity and limitations** Requiring $\pi_{exist}$ to be a subset of $\pi_{cont}$ 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 $\pi$, $\mid \pi \mid$ is be considerably close to $\mid \pi[: lca] \mid$, and the complexity of `subset()` is effectively $\mathcal{O}(2 \mid \pi \mid)$.

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 $\pi_{exist}$ and $\pi_{cont}$ contain the block of interest at some index, as can be seen in figure 3b. Consequently, $\pi_{cont}$ cannot contradict the existence of the block of interest and the predicate is evaluated *true* for both proofs. This means that if (a) $\pi_{exist}$ is structurally correct and (b) the block of interest is in $\pi_{exist}[: lca_e]$, then we can safely conclude that contesting with $\pi_{cont}$ is redundant. Therefore, $E_{contest}$ can simply send $\pi_{cont}[\text{lca}:]$ to the verifier. The truncation of $\pi_{cont}$ to $\pi_{cont}[lca_c :]$ can be easily addressed from $E_{cont}$, since $\pi_{exist}$ 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 $\pi^{tr}_{cont}$ 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.

```
1  function disjoint(
2      Proof memory exist, uint256 lca
3      Proof memory cont
4  ) internal pure returns (bool) {
5      for (uint256 i = lca+1; i < exist.length; i++) {
6          for (uint256 j = 1; j < contest.length; j++) {
7              if (exist[i] == contest[j]) { return false; }
8          }
9      }
10     return true;
11 }
```

Listing 3: Implementation for disjoint proofs

The complexity of `disjoint()` is

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

# 4 Results

# 5 Conclusion

# 6 Future Work