A gas-efficient superlight Bitcoin client in Solidity

May 3, 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

Implementation

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

1.1 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.

1.1.1 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.

1.1.2 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 1.1. Detailed gas usage of the functionalities of the verifier is shown in Table 1.1.

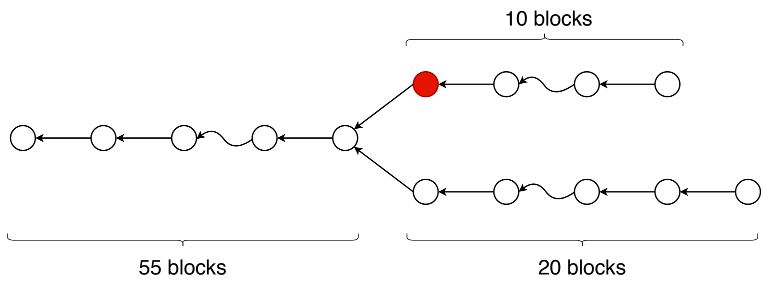


Figure 1.1: The red block indicates the block of interest. Curved connections imply intermediate blocks. The adversary creates a proof for an event that does not exist in the honest chain

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

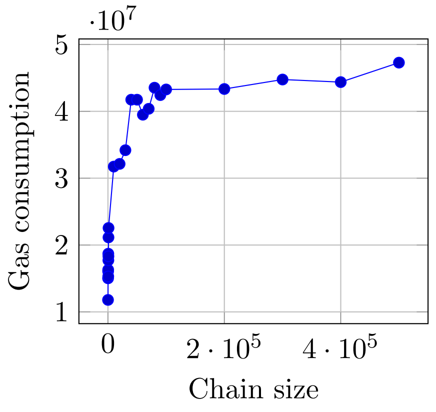
Contest function	**gas usage**
validate Interlink	213,810
find LCA	797,358
compare proofs	280,758
store proof	293,445
update DAG	1,266,348
store ancestors	3,176,599
evaluate predicate	343,056
delete ancestors	136,580
Sum	12,922,774

Table 1.1: Execution for proof of 25 blocks

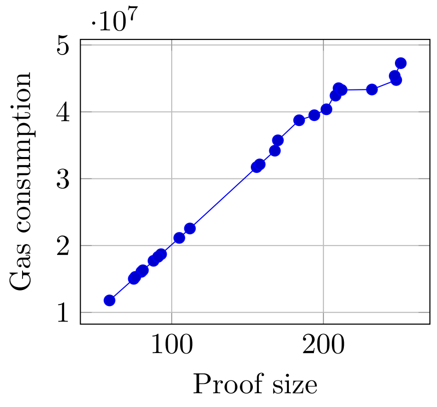
In this scenario, the original proof is created by an adversary for an event that does not exist in the honest chain. The proof is contested by an honest party. We selected this configuration because it includes both phases (submit and contest) and provides full code coverage of each phase since all 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 1.2, 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 1.2b 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 Background, 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 1.2: 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 1.2. 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 1.2: Tolls for different chain sizes. Gas price is Gwei

1.1.3 Security Analysis

We observed that previous work is vulnerable to *premining attack*. This kind of attack cannot be

Premining

Premining is the creation of a number of cryptocurrency coins before the cryptocurrency is launched to the public. Many altcoins are based on premining Bitcoin, however, is *not* a premined cryptocurrency. It is proven that every Bitcoin has been created after 3/Jan/2009, as we describe in Section **??**. If such a guarantee did not exist, the creator of Bitcoin could quietly mine blocks for a long time before initiating the public network as displayed in 1.3. The adversary could then publish the private, longer chain, invalidating the public chain which is adopted by the honest users and hosts all their funds.

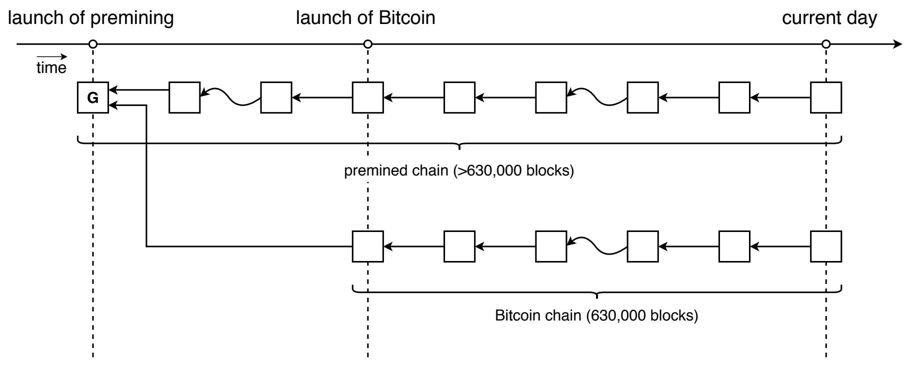

launch of premining
launch of Bitcoin
current day
time
G
premined chain (>630,000 blocks)
Bitcoin chain (630,000 blocks)

Figure 1.3: 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. As we discuss in Section **??**, in the NIPoPoW protocol a proof is structurally correct if two properties are satisfied:

Rewrite this with using the notation

(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, gen_B, results to a proof that also does not start from gen_B. Premined chains start with blocks different than gen_B, hence NIPoPoW 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' which is prior to the Bitcoin chain C as displayed in Proofs for events in C' cannot

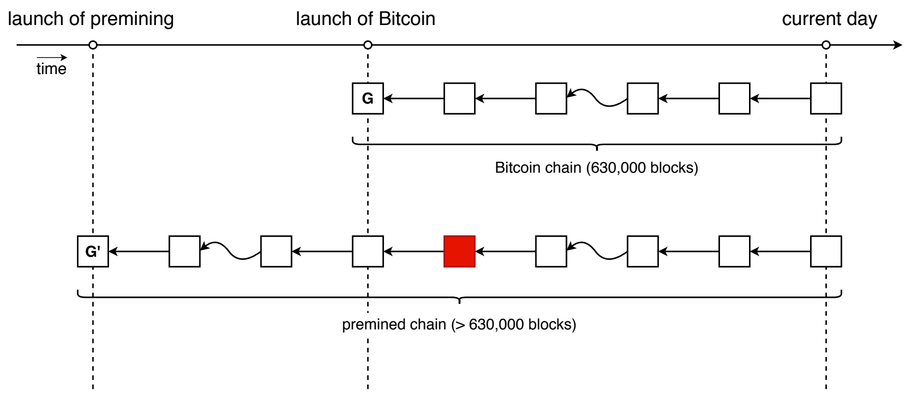

launch of premining
launch of Bitcoin
current day
time
G
Bitcoin chain (630,000 blocks)
G'
premined chain (> 630,000 blocks)

Figure 1.4: A premined chain started before Bitcoin. The older chain contains more blocks.

be contested by an honest party, since chain C' includes more proof-of-work than C and thus NIPoPoWs with higher score can be generated. Figure 1.4.

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 Algorithms 1, 2 and 3.

These operations do not noticeably increase the cost of the operations because the extra data saved in storage is constant and small in size (32 bytes).

Algorithm 1: Contract Constructor

```
  Input: $genesis$ block
1 $genesis_s \leftarrow genesis$
```

Algorithm 2: Submit Event Proof

```
Input: $\pi$, $predicate$
Data: Proof $\pi_s$, hashmap $DAG_s$, bool $predicate_s$
1 require $\pi[0] = genesis_s$
2 require $predicate_s = \emptyset$
3 require $validInterlink(\pi)$
4 $DAG_s \leftarrow DAG_s \cup \pi$
5 $\pi_s \leftarrow \pi$
6 $ancestors_s \leftarrow findAncestors(DAG_s)$
7 $predicate_s \leftarrow evaluatePredicate(ancestors_s, predicate)$
8 delete $ancestors_s$
```

Algorithm 3: Submit Contesting Proof

```
Input: $\pi'$, $predicate$
Data: Proof $\pi_s$, hashmap $DAG_s$, bool $predicate_s$
1 require $\pi'[0] = genesis_s$
2 require $predicate_s = predicate$
3 $lca \leftarrow \text{findLca}(\pi_s, \pi')$
4 require $score(proof'[lca:]) > score(proof_s[lca:])$
5 $DAG_s \leftarrow DAG_s \cup \pi'$
6 $ancestors_s \leftarrow findAncestors(DAG_s)$
7 $predicate_s \leftarrow evaluatePredicate(ancestors_s, predicate)$
8 delete $ancestors_s$
```

1.2 Logical Corrections

Tell how previous work can be improved: Predicate in contest is redundant, proof is stored only to get removed after

1.3 Storage Elimination

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.

1.3.1 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 1.1: 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 1.5. 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.

[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.

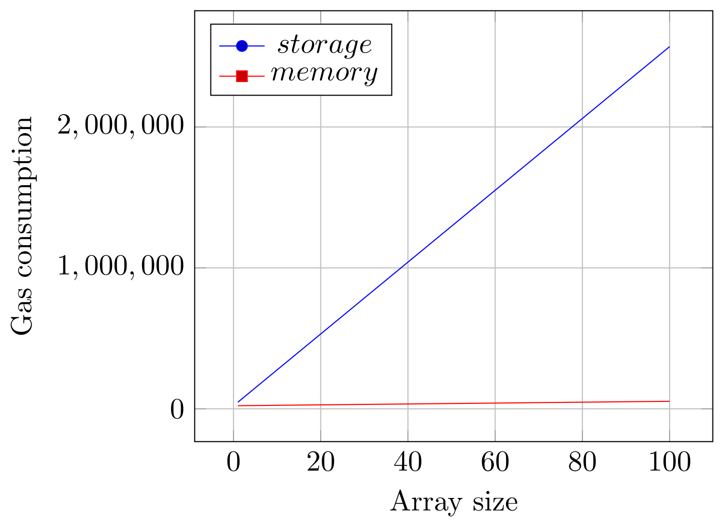


Figure 1.5: Gas consumption for memory and storage

1.3.2 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 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 1.6. 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.

1.3.3 Removing DAG and ancestors

As shown in table 1.1, 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.

[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.

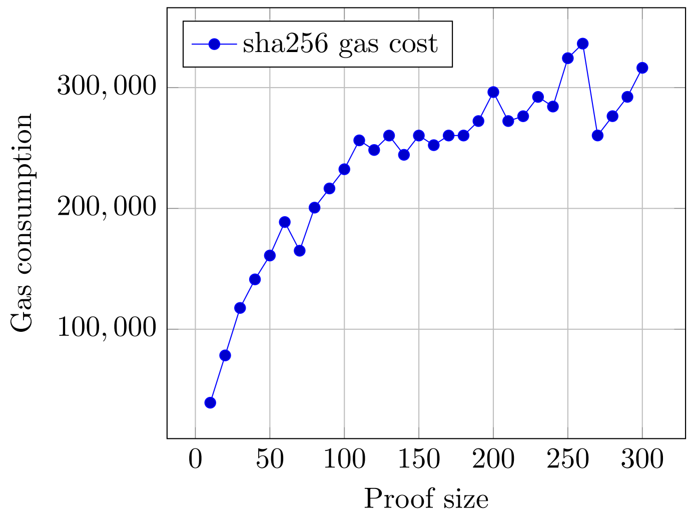


Figure 1.6: Gas consumption for hashing proofs and storing digest

Using subset

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

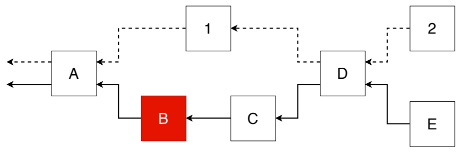
$$\pi_{exist}\{: lca_e\} \subseteq \pi_{cont}\{: lca_c\}$$

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 1.2. The complexity of the function is

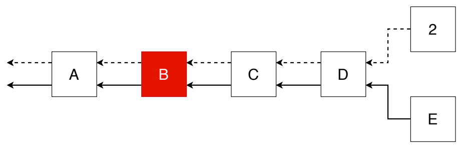
$$\mathcal{O}(\mid \pi_{exist}[: lca_e] \mid + \mid \pi_{cont}[: lca_c] \mid)$$

```
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 1.2: Implementation of subset



(a) Valid π_{cont} before using subset



(b) Valid π_{cont} after using subset

Figure 1.7: Blocks connected with solid lines indicate π_{exist} and blocks connected with dashed lines indicate π_{cont}

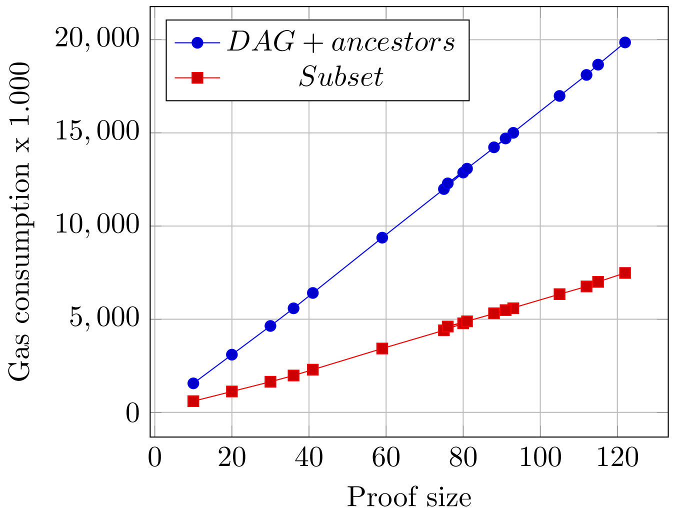


Figure 1.8: Gas consumption for DAG+ancestors and subset

The gas consumption difference between *subset* and $DAG + ancestors$ is displayed at figure 1.8. *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 $\mathcal{O}(2 \mid \pi \mid)$.

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 1.7b. 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 1.3.

```
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 1.3: Implementation for disjoint proofs

The complexity of `disjoint()` is

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

1.4 Fixing vulnerabilities and restricting gas usage