A Gas-Efficient Superlight Bitcoin Client in Solidity

Anonymous Author(s)

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. 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.

CCS CONCEPTS

• **Security and privacy** → Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper;

KEYWORDS

template; formatting; pickling

1 INTRODUCTION

Blockchain interoperability [**?**] is the ability of distinct blockchains to communicate. This *cross-chain* [**? ? ? ? ?**] communication can enable useful features across blockchains such as the transfer of asset from one chain to another (one-way peg) and back (one-way peg) [**?**]. To date, there is no commonly accepted decentralized protocol that enables cross-chain transactions. Currently, crosschain operations are only available to the users via third-party applications, such as multi-currency wallets. However, this treatment is opposing to the nature of decentralized currencies.

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

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

In order to perform crosschain operations, there must be some mechanism to allow users of blockchain A discover events that occur in chain B, such that a transaction occurred. A trivial way is to participate as a full node in both chains. This approach, however, is impractical because a sizeable amount of storage is needed to host entire chains as they grow with time. As of May 2020, Bitcoin [**?**] chain spans roughly 245 GB and Ethereum [**? ?**] has exceeded 350 GB[1]. Naturally, not all users are able to accommodate this size of data especially if portable media are used, such as mobile phones.

An early solution to compress the extensive space of blockchain was given by Nakamoto with the Simplified Payment Verification (SPV) protocol [**?**]. In SPV, only the headers of blocks are stored, saving a considerable amount of storage. However, even with this protocol, the process of downloading and validating all block headers can lead to unpleasant user experience. In Ethereum, for instance, headers sum up to approximately 5.1 GB[2] of data. A mobile client needs several minutes, even hours to fetch all information needed in order to function as an SPV client.

In order to deliver more practical solutions for blockchain verification, a new generation of *superlight* clients [**? ? ? ?**] emerged. In these protocols, cryptographic proofs are generated, which prove the occurrence of events inside a blockchain. Better performance is achieved due to considerably smaller size of proofs compared to the data needed in SPV. By utilizing superlight client protocols, a compressed proof for an event in chain A is constructed, and, if chain B supports smart contracts, the proof can then be verified automatically and transparently *on-chain*. Note that, this communication

[1]The size of the Bitcoin chain was derived from https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/, and the size of the Ethereum chain by https://etherscan.io/chartsync/chaindefaults

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

is realized without the intervention of third-party applications. An interesting application of such a protocol is the communication between Bitcoin and Ethereum.

Related Work. We use Non-Interactive Proofs of Proof of Work (NIPoPoWs) [**?**] as the fundamental building block of our solution. This cryptographic primitive is *provably secure* and provides *succinct proofs* regarding the existence of an arbitrary event in a chain. Contrary to the linear growth rate of the underlying blockchain, NIPoPoWs span polylogarithmic size of blocks.

Christoglou [**?**], 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 for their validity. This solution, however, is impossible to apply to the real blockchain due to extensive usage of resources and important security vulnerabilities such as premining.

Our contributions. We put forth the following contributions:

(1) 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. We implemented our client in Solidity, and we verify Bitcoin events to the Ethereum blockchain.
(2) We present a novel pattern which we term *hash-and-resubmit*. This pattern improves the performance of Ethereum smart contracts [**? ?**] in terms of gas consumption by utilizing *calldata* space to eliminate high-cost storage operations.
(3) We prove via application that NIPoPoWs can be utilized in the real blockchain, making the cryptographic primitive the first ever provably secure construction of succinct proofs which is applied in a real setting.
(4) Optimistic vs non-optimistic constructions - Ask gtklocker

Our implementation meets the following requirements:

(1) Security: The client is invulnerable against all adversarial attacks.
(2) Is trust-less: The client is not dependent on third-party applications and operates in a transparent, decentralized manner.
(3) 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.
(4) Is cheap: The application is cheaper to use than the current state of the art technologies.

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 and it supports smart contracts, which is a requirement in order to perform on-chain verification.

Structure. In Section 2 describe all blockchain technologies which are relevant to our work. In Section 3 we put forth .... In Section 4 we show ... and, finally, in Section 5, we discuss ...

2 THE HASH-AND-RESUBMIT PATTERN

In the Ethereum blockchain, the notion of Turing-complete smart contracts was introduced. In order to prevent accidental of adversarial denial-of-service phenomena such as infinite loops of code, contract invocations are bounded by an amount of gas units(ref). Gas consumption determines the cost a user has to pay in Ether(ref) to perform a function call, and constitutes one of the main performance criteria of smart contracts. Towards the goal of implementing gas-efficient smart contracts several patterns have been proposed.

The Pattern. We introduce a novel design pattern for Solidity smart contracts that results into massive gas optimization. This technique, which we term *hash-and-resubmit*, is based on observing public data of the blockchain in order to leverage *off-chain* operations. By utilizing *hash-and-resubmit*, the performance of the smart contract is improved considerably since computations are performed locally by the user, and gas-heavy storage variables are discarded.

The critical observation we make is that large, immutable input structures, i.e. static arrays, can be eliminated. This is due to the fact that in the body of the Ethereum block, information regarding transactions is stored. Nodes have access to this information which include the signatures and input arguments of function invocations. Hence, a node can access the information of a transaction by reading the blockchain instead of invoking contract functions. This type of off-chain access can be applied by any node.

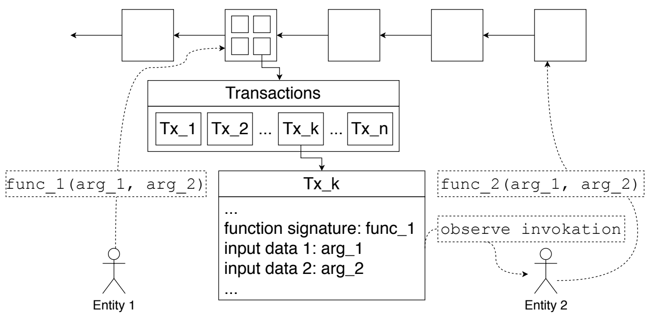

Figure 1: Entity 1 makes a function call with arguments arg1 and arg2. Entity 2 reads the content of transactions in the block and retrieves the input data. Entity 2 can use this data to make a different invocation.

As an example, we demonstrate a smart contract that implements a game between two players, $P1$ and $P2$, in which the player with the most valuable array wins. The contract consists of two phases: (a) Submit phase and (b) Contest phase.

Algorithm 1 Compare N-sized arrays using storage

```
function Submit($array$)
    $best\_array \leftarrow array$                ▷ array saved in storage
    $holder \leftarrow msg.sender$
    return true
end function

function Contest($new\_array$)
    if !compare($new\_array$) then
        return false
    end if
    $holder \leftarrow msg.sender$
    return true
end function

function compare($new\_array$)
    for $i$ in $best\_array$.length do
        if $new\_array[i] \leq best\_array[i]$ then
            return false
        end if
    end for
    return true
end function
```

Algorithm 2 Compare N-sized arrays using hash-and-resubmit pattern

```
function Submit($array[N]$)
    $hash \leftarrow sha256(array)$             ▷ hash saved in storage
    $holder \leftarrow msg.sender$
    return true
end function

function Contest($existing\_array$, $new\_array$)
    if hash256($existing\_array$) $\neq hash$ then
        return false                      ▷ Invalid $original\_array$
    end if
    if !compare($original\_array$, $new\_array$) then
        return false
    end if
    $holder \leftarrow msg.sender$
    return true
end function

function compare($array_1$, $array_2$)
    for $i$ in $array_1$.length do
        if $array_1[i] \leq array_2[i]$ then
            return false
        end if
    end for
    return true
end function
```

Submit phase: $P1$ submits an N-sized array, $array_1$ and becomes the *holder* of the contract.
Contest phase: $P2$ submits $array_2$. If $array_2 > array_1$, then the holder of the contract is changed.

In this example, the comparison $array_1 > array_2$ is true if $array_1[i] > array_2[i]$ is true for every $i \in array_1$.length. However, the pattern is agnostic to the implementation of the operator.

The rationale is to demand from $P2$ to provide two arrays to the contract during contest phase: (a) $array_1^*$, which is a copy of the originally submitted array by $P1$, and (b) $array_2$, which is the contesting array. The challenge here, is twofold:

(1) Availability: $P2$ must be able to retrieve a valid copy of $array_1$, without the need of accessing on-chain data.
(2) Reliability: We must prevent $P2$ from dispatching a malformed $array_1^*$ which differs from the originally submitted $array_1$.

As its name suggests, *hash-and-resubmit* technique is performed in two stages to face these challenges: (a) the *hash* phase, which addresses *reliability*, and (b) the *resubmit* phase which addresses *availability.*
Addressing Availability: During submit-phase, $P1$ make the function call `submit`($array_1$). This transaction, which includes the function signature (`submit`) and the corresponding data ($array_1$), is added to a block by a miner. Due to blockchain's transparency, the observer of `submit`, $P2$, retrieves a copy of $array_1$, without the need of accessing contract data. We denote as the copy of $array_1$ as $array_1^*$.
Addressing Reliability: We prevent an adversary from altering $array_1^*$ by storing the hash of $array_1$ in contract's state during *submit phase.* We make use of the pre-compiled `sha256` hash function of Solidity:

$$hash = \texttt{sha256}(array_1)$$

Then, during contest phase, a verification is performed by comparing the stored hash of $array_1$ against the hash of $array_1^*$.

$$\texttt{require}(hash = \texttt{sha256}(array_1^*))$$

The size of the hash is 32 bytes. To persist such a small value in contract's memory only adds a constant, negligible cost overhead.
Benchmarks. We show the storage implementation of the above game in Algorithm **??** and the hash-and-resubmit version in Algorithm **??**. The gas consumption of the two algorithms is displayed in Figure 2.

By using the hash-and-resubmit, the gas consumption decreased by 93-95%. This significantly affects the applicability of the contract. Note that, the storage implementation exceeds the Ethereum block gas limit[3] for arrays of size 500 and above, contrary to the hash-and-resubmit-optimized version, which consumes approximately 1/10th of the block gas limit for arrays of 1000 elements.

[3] As of July 2020, the Ethereum block gas limit approximates 10,000,000 gas units

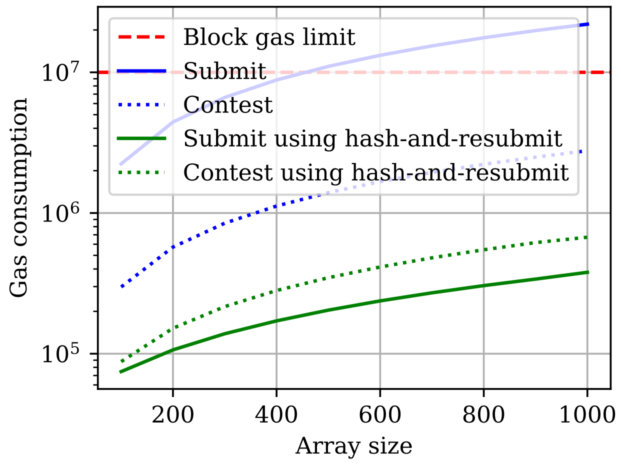

Figure 2: Gas-cost reduction using the *hash-and-resubmit* pattern. By avoiding gas-heavy storage operations, the cost of function invocations is decreased significantly by 93-95%.

Enabling NIPoPoWs. We now present how the *hash-and-resubmit* pattern can used in the context of the NIPoPoW superlight client. Similar to the aforementioned example, the NIPoPoW verifier adheres to a submit-and-contest-phase schema, and the inputs of the functions are arrays that are processed on-chain.

In *submit-phase*, a *proof* is submitted, which can be contested by another user in *contest-phase*. The user that initiates the contest, needs to monitor the traffic of the smart contract. This is a logical assumption as mentioned in the NIPoPoW paper. The input of `submit` function includes the *proof*, π_{subm} that indicates the occurrence of an *event*, **e** in the source chain, and the input of `contest` function includes the *contesting proof*, π_{cont}. A successful challenge of the π_{subm} is realized when π_{cont} has better score. The process of score evaluation, which is described in [**?**] is irrelevant to the pattern, and remains unchained.

In previous work [**?**], NIPoPoW arrays are stored on-chain during submit-phase, and are processed during contest-phase. This operation is performed by utilizing storage, which limits the applicability of the contract considerably. In Algorithm **??** we show how hash-and-resubmit pattern can be embedded into the NIPoPoW client. In Figure 3 we display how the gas consumption differentiates from previous work for the aggregated cost of submit and contest phase.

3 REMOVING MUTABLE STORAGE

Now that we can freely retrieve immutable structures, we can focus on other storage variables. A challenge we faced is that the protocol of NIPoPoWs dependents on *DAG* structure which is a mutable hashmap. This logic is intuitive and efficient to implement in most traditional programming

Algorithm 3 The NIPoPoW client using hash-and-resubmit pattern

```
1:  function Constructor($genesis$)
2:      $Gen \leftarrow genesis$
3:  end function

1:  function Submit($\pi$, $e$)
2:      require($\pi[0] = Gen$)
3:      require($events[e].pred$ = false)
4:      require($validInterlink(\pi)$)
5:      $DAG_s \leftarrow DAG_s \cup \pi$
6:      $events[e].hash \leftarrow$ sha256($\pi$)            ▷ enable pattern
7:      $ancestors_s \leftarrow findAncestors(DAG_s)$
8:      $events[e].pred \leftarrow evaluatePredicate(ancestors_s, e)$
9:      delete $ancestors$
10: end function

1:  function Contest($\pi$, $\tilde{\pi}$, $e$)    ▷ dispatch $\pi^*_{sumb}$ and $\pi_{cont}$
2:      require($events[e].hash$ = sha256($\pi$))   ▷ verify $\pi^*_{subm}$
3:      require($\tilde{\pi}[0] = Gen$)
4:      require($events[e].pred$ = true)
5:      require($validInterlink(\tilde{\pi})$)
6:      $lca = findLca(\pi, \tilde{\pi})$
7:      require($score(\tilde{\pi}[: lca]) > score(\pi[: lca])$)
8:      $DAG \leftarrow DAG \cup \tilde{\pi}$
9:      $ancestors_s \leftarrow findAncestors(DAG_s)$
10:     $events[e].pred \leftarrow evaluatePredicate(ancestors_s, e)$
11:     delete $ancestors$
12: end function
```

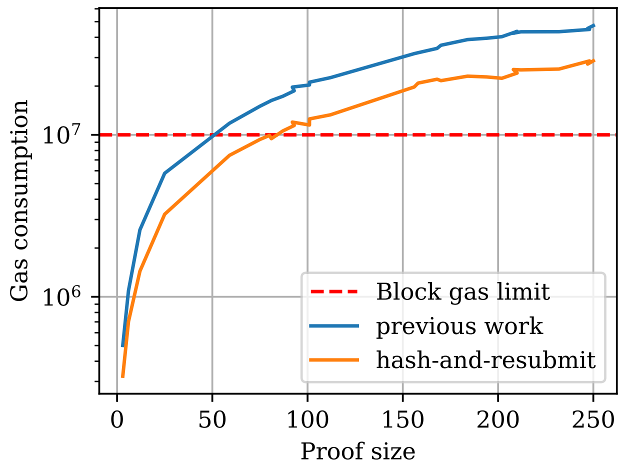

Figure 3: Performance improvement using hash-and-resubmit pattern in NIPoPoWs related to previous work. The gas consumption decreased by approximately 50%

languages such as C++, JAVA, Python, JavaScript, etc. However, as our analysis demonstrates, such an algorithm cannot be efficiently implemented in Solidity. This is not due to the lack of support of look-up structures, but because Solidity hashmaps can only be contained in storage. Other mutable structures, such as *ancestors* that are stored in the persistent memory also affect performance. Thus, the use of *DAG* leads to expensive function calls, especially for large proofs.

We make a keen observation regarding the possible position of the *block of interest* in the proof which lead us the constriction of an architecture that does not require *DAG*, *ancestors* or any other complementary structures. We will use the notation from [**?**] to present our claim.

Position of block of interest. NIPoPoWs are sets of sampled interlinked blocks, which means that they can be perceived as chains. If π_1 differs from π_2, then a fork is created at the index of the last common ancestor (lca). The block of interest, b, lies at a certain index within π_{subm} and indicates a stable predicate [**? ?**] that is true for π_{subm}. We do not consider the absence of b from π_{subm}, because in this case the submission automatically fails since the predicated is evaluated against π_{subm}. On the contrary, the entity that initiates the contest of π_{subm}, tries to prove the falseness of the underlying predicate against π_{cont}. This means that, if the block of interest is included in π_{cont}, the contest is aimless. We will refer to such aimless actions as *irrational* and components that are included in such actions irrational components, i.e. irrational proof, blocks etc, and we will use the term *rational* to describe non-irrational actions and components.

In the NIPoPoW protocol, proofs' segments $\pi_{subm}\{: lca\}$ and $\pi_{cont}\{: lca\}$ are merged to prevent adversaries from skipping or adding blocks, and the predicate is evaluated against $\pi_{subm}\{: lca\} \cup \pi_{cont}\{: lca\}$. We observe that $\pi_{submit}\{: lca\}$ can be omitted because there is no block $\{B : B \notin \pi_{subm}\{: lca\} \wedge B \in \pi_{cont}\{: lca\}\}$ that results to positive evaluation of the predicate. This is due to the fact that b is not included in $\pi_{cont}[: lca]$, and, presumably there is no B such that $b = B$.

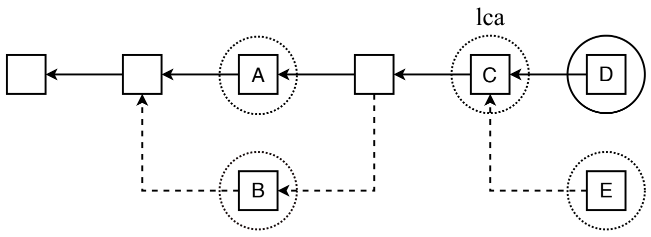

Figure 4: Fork of two chains. Solid lines connection blocks of π_{subm} and dashed lines connect blocks of π_{cont}. Given the configuration, blocks in dashed circles are irrational blocks of interest, and the block i in the solid circle is a rational block of interest.

In Figure 4 we display a fork of two proofs. Solid lines connect blocks of π_{subm} and dashed lines connect blocks of π_{cont}. Examining which scenarios are rational depending on different positions of the block of interest, we observe that blocks `B`, `C` and `E` do not qualify, because they are included only in π_{cont}. Block `A` is included in $\pi_{subm}\{: lca\}$, which means that π_{cont} is an irrational contest. Given this configuration, the only rational block of interest is `D`.

Minimal forks. By combining the above observations, we derive that, π_{cont} can be truncated into $\pi_{cont}\{: lca\}$ without affecting the correctness of the protocol. We will term this truncated proof π^{tr}_{cont} [4]. Security is preserved if we require π^{tr}_{cont} to be a *minimal fork* of π_{subm}. A minimal fork is a fork chain that shares exactly one common block with the main chain. Proof $\tilde{\pi}$, which is a minimal fork of proof π, has the following attributes:

(1) $\pi\{lca\} = \tilde{\pi}[0]$
(2) $\pi\{lca :\} \cap \tilde{\pi}[1 :] = \emptyset$

By requiring that π^{tr}_{cont} is a minimal fork of π_{subm}, we prevent an adversary from dispatching an augmented π^{tr}_{cont} to claim better score against π_{subm}. Such an attempt is displayed in Figure 5.

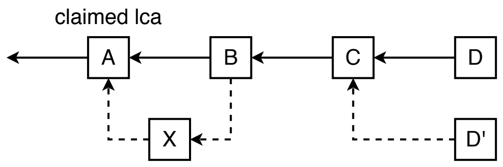

Figure 5: An adversary tries to send a malformed proof consisting of blocks {A, X, B, C, D'} that achieves better score against {A, B, C, D}. This attempt is rejected due to the minimal-fork requirement.

In Algorithm **??**, we show how minimal fork technique is incorporated in our client, replacing *DAG* and *ancestors*. In Figure 6 we show how the performance of the client has improved.

[4]We cannot proceed to further truncation of π^{tr}_{cont}, because in the NIPoPoW protocol blocks within segment $\pi\{: lca\}$ of each proof are required for the score calculation.

Algorithm 4 The NIPoPoW client using the minimal fork technique

```
1: function Submit($\pi$, $e$)
2:     require($\pi[0] = Gen$)
3:     require($events[e]$ = false)
4:     require($validInterlink(\pi)$)
5:     $events[e].pred \leftarrow evaluatePredicate(\pi, e)$
6:     $events[e].hash \leftarrow$ sha256($\pi$)
7: end function

1: function Contest($\pi$, $\tilde{\pi}$, $e$, $f$)          ▷ $f$: fork index in $\pi$
2:     require($\tilde{\pi}[0] = \pi[f]$)          ▷ check min. fork head
3:     require($events[e].hash$ = sha256($\pi$))
4:     require($events[e].pred$ = true)
5:     require($validInterlink(\tilde{\pi})$)
6:     require($disjoint(\pi[f+1:], \tilde{\pi}[1:])$)  ▷ check min. fork
7:     require($score(\tilde{\pi}) > score(\pi[f:])$)
8:     $events[e].pred \leftarrow evaluatePredicate(\tilde{\pi}, e)$
9: end function
```

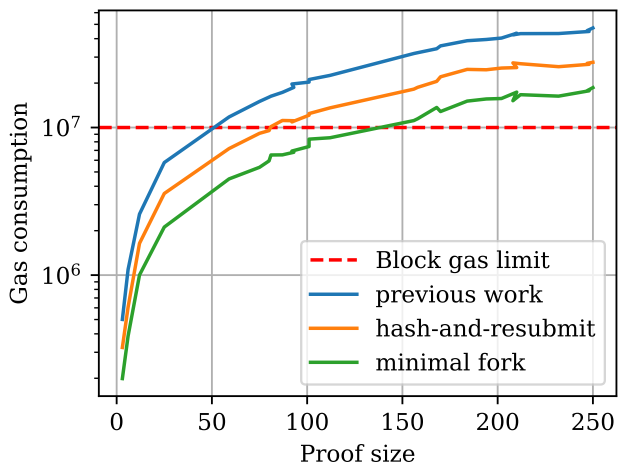

Figure 6