

RePoPoW: A Gas-Efficient Superlight Bitcoin Client in Solidity

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.

KEYWORDS

Blockchain; Superlight clients; NIPoPoWs, Solidity

1 INTRODUCTION

Blockchain interoperability [19] is the ability of distinct blockchains to communicate. This *crosschain* [8, 10, 11, 14, 18] communication enables useful features across blockchains such as the transfer of asset from one chain to another (one-way peg) and back (one-way peg) [14]. 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 support the following operations:

- Crosschain trading: A user with deposits in blockchain A, makes a payment to a user in blockchain B.
- Crosschain fund transfer: A user transfers her funds from blockchain A to blockchain B. After the transfer, these 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 a mechanism to allow for users of blockchain A to discover events that occur in chain B, such that a transaction occurred. A trivial manner to perform such an audit is to participate as a full node in multiple 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 July 2020, Bitcoin [16] chain spans roughly 245 GB, and Ethereum [3, 17] has exceeded 350 GB¹. Naturally, not all users are able to accommodate this size of data, especially if portable media are used, such as mobile phones.

One early solution to compress the extensive size of blockchain is addressed by Nakamoto [16] 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 leads to unpleasant user experience. In Ethereum, for instance, headers sum up to approximately 5.1 GB² of data. A mobile client needs several minutes, even hours, to fetch all information needed in order to function as an SPV client.

Towards the goal of delivering more practical solutions for blockchain transaction verification, a new generation of *superlight* clients [2, 9, 12, 13] emerged. In these protocols, cryptographic proofs are generated, that prove the occurrence of events inside a blockchain. Better performance is achieved due to the considerably smaller size of proofs compared to the amount of data needed in SPV. By utilizing superlight client protocols, a compressed proof for an event in chain A is constructed. Under the consumption that chain B supports smart contracts, the proof is then verified automatically and transparently *on-chain*. This communication is realized without the intervention of third-party applications. An interesting application of such a protocol is the communication between Bitcoin and Ethereum.

Notably, no practical implementation exists for superlight clients to date. In this paper, we focus on constructing a

¹The 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>

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

practical client for Non-Interactive Proofs of Proof of Work (NIPoPoWs). In the process of implementing our client, we refine the NIPoPoW protocol based on a series of keen observations. These refinements allow us to leverage useful techniques that construct a practical solution for proof verification. We believe that this achievement is a decisive step towards the establishment of NIPoPoWs to the application end, therefore a significant progress in order to provide a widely accepted protocol that enables crosschain transactions.

Related Work. We use Non-Interactive Proofs of Proof of Work (NIPoPoWs) [13, 14] 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 [6] 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 due to extensive usage of resources, widely exceeding the Ethereum block gas limit.

Our contributions.

We put forth the following contributions:

- (1) We developed the first decentralized client that securely verifies crosschain events and is practical. Our client establishes a trustless and efficient solution to the interoperability problem. We implement 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*. Our pattern significantly improves performance of Ethereum smart contracts [3, 17] in terms of gas consumption by utilizing *calldata* space of Ethereum blockchain to eliminate high-cost storage operations.
- (3) We design an *optimistic* schema for our client which we incorporate in the design of our client, replacing *non-optimistic* architectures. This design achieves the improvement of smart contracts' performance enabling multiple phases of interactions.
- (4) We prove via application that the NIPoPoW protocol is practical, making the cryptographic primitive the first provably secure construction of succinct proofs that is efficient to implement.
- (5) Cryptoeconomics. Create the section first.

Our implementation meets the following requirements:

- (1) Security: The client implements a provably secure protocol.
- (2) Decentralization: The client is not dependent on third-party applications and operates in a transparent, decentralized manner.
- (3) Efficiency: The client comply with all environmental constraints, i.e. block gas limit and *calldata* size limit of Ethereum blockchain.

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

Structure. In Section 2 we describe the blockchain technologies that are relevant to our work. In Section 3 we put forth the *hash-and-resubmit* pattern. We demonstrate the improved performance of smart contracts using the pattern, and how it is incorporated in our superlight client. In Section 4 we show ... and. Finally, in Section 5, we discuss ...

2 PRELIMINARIES

3 THE HASH-AND-RESUBMIT PATTERN

We now introduce a novel design pattern for Solidity smart contracts that results into massive gas optimization due to the elimination of expensive storage operations.

Motivation. It is essential for smart contracts to store data in the blockchain. However, interacting with the storage of a contract is among the most expensive operations of the EVM [3, 17]. Therefore, only necessary data should be stored and redundancy should be avoided when possible. This is contrary to conventional software architecture, where storage is considered cheap. Usually, the performance of data access in traditional systems is related with time. In Ethereum, however, performance is related to gas consumption. Access to persistent data costs a substantial amount of gas, which has a direct monetary value. One way to mitigate gas cost of reading variables from the blockchain is to declare them public. This leads to the creation of a *getter* function in the background, allowing free access to the value of the variable. But this treatment does not prevent the initial population of storage data, which is significantly expensive for large size of data. Towards the goal of implementing gas-efficient smart contracts, several patterns have been proposed [4, 5, 7].

By using the *hash-and-resubmit* pattern, large storage structures are omitted entirely, and are contained in memory which results into vastly improved performance. When a function call is performed, the arguments and signature of the function is included in the transactions field of the body of a block. The contents of blocks are public to the network, therefore this information is locally available to full nodes. By simply observing blocks, a node retrieves data sent by other users. To interact publicly with the data sent by other users without the utilization of on-chain storage, the node *resends* the observed data to the blockchain. Consequently, this data does not need to be maintained in storage. The concept of resending data would be redundant in conventional systems. However, it is very efficient to utilizing the technique in Solidity because it leverages operations in memory rather than storage, which are notable cheaper.

Applicability. We now list the cases in which the *hash-and-resubmit* pattern is efficient to use:

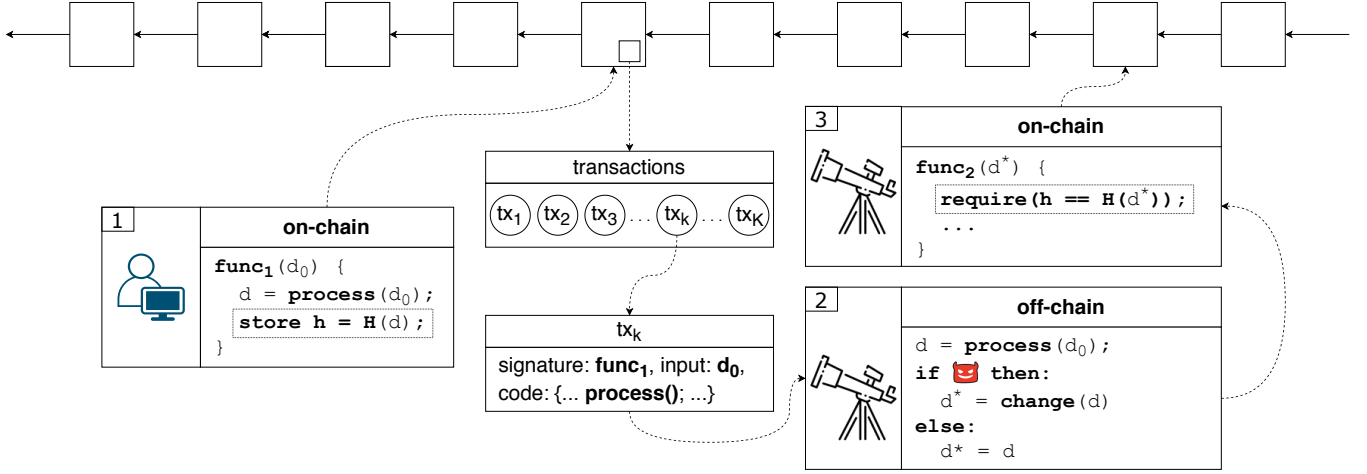


Figure 1: The *hash-and-resubmit* pattern. In Stage 1, an invoker calls `func1(d0)`. `d0` is processed on-chain and `d` is generated. The signature of `d` is stored in the blockchain as the digest of a hash function `H()`. In Stage 2, a full node that observes invocations of `func1` retrieves `d0`, and generates `d` by performing the analogous processing on `d0` off-chain. An adversarial observer potentially alters `d`. Finally, in Stage 3, the observer invokes `func2(d*)`. In `func2`, the validation of `d` is performed, reverting the function call if the signatures of originally submitted `d` does not match the signature of `d*`. By applying the *hash-and-resubmit* pattern, only fixed-size signatures of data need to be maintained on the blockchain replacing arbitrarily large structures.

- (1) To reduce gas inefficiencies caused by extensive read/write storage operations and to make smart contracts that exceed block gas limit practical.
- (2) To interact with smart contract depending on prior actions of other users.
- (3) To leverage off-chain operations.

Participants and collaborators. The first participant is the smart contract `S` that accepts function calls. Another participant is the invoker `E1`, who dispatches arbitrary data `d0` to `S` via a function `func1(d0)`. Note that `d0` are potentially processed in `func1`, resulting to `d`. The last participant is the observer `E2`, who is a full node that observes transactions towards `S` in the blockchain. This possible because nodes maintain the blockchain locally. After observation, `E2` retrieves data `d`. Finally, `E2` acts as an invoker by making a new interaction with `S`, `func2(d)`. Since this is an off-chain operation, a malicious `E2` can alter `d` before interacting with `S`. We will denote the potentially modified `d` as `d*`. The verification that `d = d*`, which is a prerequisite for the secure functionality of the underlying contract consists a part of the pattern and is performed in `func2(d*)`.

Implementation. The implementation of this pattern is divided in two parts. The first part covers how `d*` is retrieved by `E2`, whereas in the second part the verification of `d = d*` is realized. The challenge here is twofold:

- (1) Availability: `E2` must be able to retrieve `d` without the need of accessing on-chain data.
- (2) Consistency: `E2` must be prevented from dispatching `d*` that differs from the originally submitted `d`.

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

Addressing availability: During *hash* phase, `E1` makes the function call `func1(d0)`. This transaction, which includes a function signature (`func1`) and the corresponding data (`d0`), is added in a block by a miner. Due to blockchain's transparency, the observer of `func1`, `E2`, retrieves a copy of `d0`, without the need of accessing contract data. In turn, `E2` performs *locally* the same set of on-chain instructions operated on `d0` generating `d`. Thus, availability is addressed through observability.

Addressing reliability: We prevent an adversary `E2` from altering `d*` by storing the *signature* of `d` in contract's state during the execution of `func1(d)` by `E1`. In the context of Solidity, a signature of a structure is the digest of the structures's *hash*. The pre-compiled `sha256` is convenient to use in Solidity, however we can make use of any cryptographic hash function `H()`:

$$\text{hash} \leftarrow H(d)$$

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

$$\text{require}(\text{hash} = H(d^*))$$

In Solidity, the size of digests is 32 bytes. To persist such a small value in contract's memory only adds a constant, negligible cost overhead.

We illustrate the application of the *hash-and-resubmit* pattern in Figure 1.

Sample. We now demonstrate the usage of the hash-and-resubmit pattern with a practical example. We create a smart contract that orchestrates a game between two players, P_1 and P_2 . The winner is the player with the most valuable array. The interaction between players through the smart contract is realized in two phases: (a) Submit phase and (b) Contest phase.

Submit phase: P_1 submits an N -sized array, a_1 and becomes the holder of the contract.

Contest phase: P_2 submits a_2 . If $a_2 > a_1$, then the holder of the contract is changed to P_2 . We provide a simple implementation for the $>$ operator between arrays, but we can consider any notion of comparison between arrays, since the pattern is abstracted from such implementation details.

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

We provide two implementations of the above described game. Algorithm ?? is the storage implementation and Algorithm ?? is the implementation embedding the *hash-and-resubmit* pattern.

Algorithm 1 best array using storage

```

1: contract best-array
2:   function initialize
3:     best  $\leftarrow \emptyset$ ; holder  $\leftarrow \emptyset$ 
4:   end function
5:   function submit( $a$ )
6:     best  $\leftarrow a$  ▷ array saved in storage
7:     holder  $\leftarrow$  msg.sender
8:   end function
9:   function contest( $a$ )
10:    require(compare( $a$ ))
11:    holder  $\leftarrow$  msg.sender
12:  end function
13:  function compare( $a$ )
14:    require(| $a$ |  $\geq$  |best|)
15:    for  $i : |best|$  do
16:      if  $a[i] \leq \text{best}[i]$  then return false
17:    end if
18:  end for
19:  return true
20: end function
21: end contract

```

Gas analysis. The gas consumption of the two implementations is displayed in Figure 2. By using the *hash-and-resubmit* pattern, the overall gas consumption for **submit** and **contest** is decreased by 95%. This significantly affects the efficiency and applicability of the contract. Note that, the storage implementation exceeds the Ethereum block gas limit³ for arrays

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

Algorithm 2 best array using hash-and-resubmit pattern

```

1: contract best-array
2:   function initialize
3:     hash  $\leftarrow \emptyset$ ; holder  $\leftarrow \emptyset$ 
4:   end function
5:   function submit( $a_1$ )
6:     hash  $\leftarrow H(a_1)$  ▷ hash saved in storage
7:     holder  $\leftarrow$  msg.sender
8:   end function
9:   function contest( $a_1^*, a_2$ )
10:    require(hash256( $a_1^*$ ) = hash) ▷ validate  $a_1^*$ 
11:    require(compare( $a_1^*, a_2$ ))
12:    holder  $\leftarrow$  msg.sender
13:  end function
14:  function compare( $a_1^*, a_2$ )
15:    require(| $a_1^*$ |  $\geq$  | $a_2$ |)
16:    for  $i : |a_1^*|$  do
17:      if  $a_1^*[i] \leq a_2[i]$  then return false
18:    end if
19:  end for
20: end function
21: return true
22: end contract

```

of size 500 and above, contrary to the optimized version, which consumes approximately only $1/10^{th}$ of the block gas limit for arrays of 1000 elements.

Variations. Now consider a variation of the above game, in which P_1 calls **func₁**(a_1), and then calls **pickSpan**(m, n) that determines the span of a_1 which can be contested. In reality, P_2 only needs to re-send $a_1^*[m : n]$ in order to perform the comparison $a_1[m : n] < a_2$. However, the digest of a_1 is calculated by hashing the entire structure. Therefore, the *resubmit* phase cannot be successfully performed by rehashing $a_1^*[m : n]$, because $H(a_1) \neq H(a_1[m : n])$.

An intuitive approach to address such scenarios in order to facilitate selective dispatch of structure segments is to adopt different hashing schemas that utilize constructions such as Merkle Trees (ref) or Merkle Mountain Ranges (ref). We will proceed by only mentioning Merkle Trees, but since the respective operations for proving and validating roots is of the same complexity, the same principle applies for Merkle Mountain Ranges.

In the Merkle variation of the pattern, which we term *merkle-hash-and-resubmit*, the signature of a_1 is generated by constructing the Merkle Tree Root (MTR) of a_1 in contract's state. In *resubmit* phase, $a_1[m : n]$ is dispatched, accompanied by the siblings that reconstruct the MTR of a_1 in order to perform the variation of $a_1^*[m : n]$.

This variation of the pattern removes the burden of sending redundant data, however it implies on-chain construction and validation of the Merkle construction. In order to construct a MTR for an array a of size n , the hash function $H(|a[n]|)$ needs to be called approximately $2^{\lceil \log_2(n) + 1 \rceil}$ times. In order to reconstruct the Merkle Tree Root for the verification,

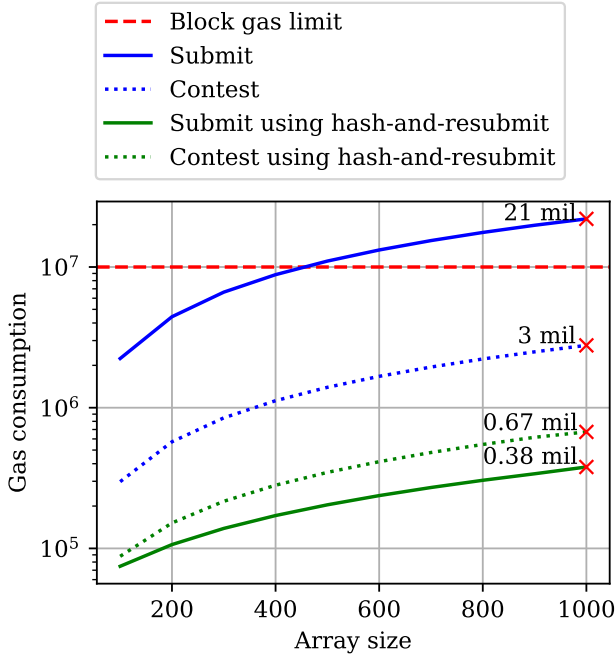


Figure 2: Gas-cost reduction using the *hash-and-resubmit* pattern. By avoiding gas-heavy storage operations, the aggregated cost of submit and contest is decreased significantly by 95%.

approximately $\lceil \log_2 n \rceil$ calls to $H(\cdot)$ are needed. The process of constructing and verifying the MTRoot is displayed in Figure 3. Note that both operations need to be performed on-chain.

In Solidity, hashing operations are expensive compared to memory cost. An invocation of `sha256(a)`, copies the data in memory, and then a `CALL` instruction is performed by the EVM to the pre-compiled `sha256` contract. In the current state of the EVM, `CALL` costs 700 gas units, and the gas paid for every word when expanding memory is 3 gas units [17]. Costs of all related operations are listed in Table 1. Due to this large discrepancy in gas costs, it is extremely more efficient to send data of size $|a|$ and perform $1 \times \text{sha256}(a)$ than sending data of size $\log_2(|a|)$ and perform $2^{\lceil \log_2(|a|) \rceil} \times \text{sha}(1)$ operations. A different cost policy applies for `keccak` [1] hash function, where hashing costs 30 gas units plus 6 additional gas far each word for input data. Although the usage of `keccak` dramatically increases the performance in comparison with `sha256`, plain rehashing generally performs better.

In Table 2 we display the operations needed for hashing and verifying the underlying data for both variations of the pattern as a function of data size. In Figure 4 we demonstrate the gas consumption for $d_0 = 10\text{KB}$ and d varying from $0.01 \times |d_0|$ to $2 \times |d_0|$.

Operation	Gas cost
<code>load(d)</code>	$68 \times d $
<code>mem(d)</code>	$4 \times d $
<code>sha256(d)</code>	$4 \times d + 700$
<code>keccak(d)</code>	$6 \times d + 30$

Table 1: Gas cost for operations as of July 2020.

actions	hash and resubmit	merkle hash and resubmit
<i>hash</i>	$H(d)$	$\text{load}(d) + H(1) \times 2^{\lceil \log_2(d +1) \rceil}$
<i>resubmit</i>	$\text{load}(d) + H(d)$	$\text{load}(d^{m:n}) + \text{load}(\lceil \log_2(d) \rceil) + H(1) \times (\lceil \log_2(d) \rceil)$

Table 2: Summary of operations for *hash-and-resubmit* pattern variations. d is the product of on-chain operations. In the Merkle variation, a Merkle Tree Root is constructed on-chain during *hash* phase. In *resubmit* phase, the siblings of the Merkle Tree are dispatched, and the verification is performed on-chain. The number of siblings is approximately $\lceil \log_2(|d|) \rceil$.

Consequences. The most obvious consequence of applying the *hash-and-resubmit* pattern variations is the circumvention of storage structures, a benefit that saves a substantial amount of gas, especially in the cases where these structures are large. To that extend, smart contracts that exceed the Ethereum block gas limit become practical. Furthermore, the pattern enables off-chain transactions, significantly improving the performance of smart contracts.

Known uses. To our knowledge, we are the first to combine the notion of the transparency of the blockchain with data structures signatures to eliminate storage variables from Solidity smart contracts by resubmitting data.

Enabling NIPoPoWs. We now present how the *hash-and-resubmit* pattern can be 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, monitors the traffic of the smart contract [13]. The input of *submit* function includes the submit 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 contest of π_{subm} is realized when π_{cont} has a better score. The score evaluation process is irrelevant to the pattern and remains unchained. The size of proofs is dictated by the value m . We consider $m = 15$ sufficiently secure.

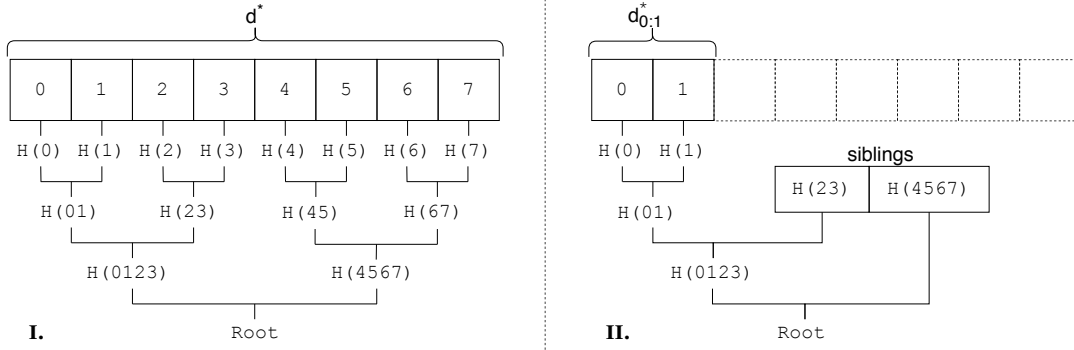


Figure 3: I. The calculation of root in *hash* phase. II. The verification of the root in *resubmit* phase. $H(k)$ denotes the digest of element k . $H(kl)$ denotes the result of $H(H(k) \parallel H(l))$

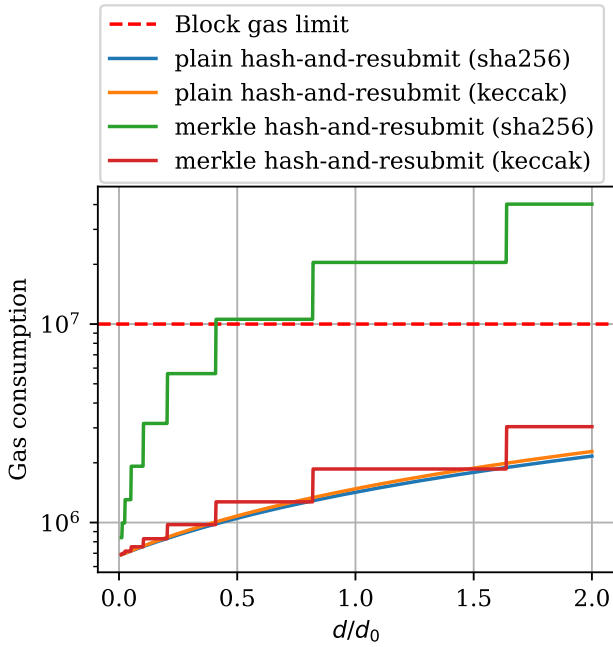


Figure 4: Trade-offs between *hash-and-resubmit* variations. In the vertical axis the gas consumption is displayed, and in vertical axis the size of d as a function of d_0 . The size of d_0 is 10KB bytes, and the hash function we used is pre-compiled sha256.

In previous work [6], NIPoPoW proofs are maintained on-chain, resulting to extensive storage operations that limit the applicability of the contract considerably. In Algorithm ?? we show how hash-and-resubmit pattern is embedded into the NIPoPoW client. In Figure 5, we display how results of the *hash-and-resubmit* implementation differentiate from previous work for the aggregated cost of *submit* and *contest* phases. We observe that by using the *hash-and-resubmit* pattern,

we achieve to increase the performance of the contract 40%. This is a decisive step towards creating a practical superlight client.

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

```

1: contract crosschain
2:   function initialize( $G_{remote}$ )
3:      $G \leftarrow G_{remote}$ 
4:   end function
5:   function Submit( $\pi_{subm}, e$ )
6:     require( $\pi_{subm}[0] = G$ )
7:     require( $events[e] = \perp$ )
8:     require(validInterlink( $\pi$ ))
9:      $DAG \leftarrow DAG \cup \pi_{subm}$ 
10:     $events[e].hash \leftarrow H(\pi_{subm})$   $\triangleright$  enable pattern
11:     $ancestors \leftarrow findAncestors()$ 
12:     $events[e].pred \leftarrow evaluatePredicate(ancestors, e)$ 
13:     $ancestors = \perp$ 
14:  end function
15:  function Contest( $\pi_{subm}^*, \pi_{cont}, e$ )  $\triangleright$  provide proofs
16:    require( $events[e].hash = H(\pi_{subm}^*)$ )  $\triangleright$  verify  $\pi_{subm}^*$ 
17:    require( $\pi_{cont}[0] = G$ )
18:    require( $events[e] \neq \perp$ )
19:    require(validInterlink( $\pi_{cont}$ ))
20:     $lca = findLca(\pi_{subm}^*, \pi_{cont})$ 
21:    require( $score(\pi_{cont}[lca]) > score(\pi_{subm}^*[lca])$ )
22:     $DAG \leftarrow DAG \cup \pi_{cont}$ 
23:     $ancestors \leftarrow findAncestors(DAG)$ 
24:     $events[e].pred \leftarrow evaluatePredicate(ancestors, e)$ 
25:     $ancestors = \perp$ 
26:  end function
27: end contract

```

4 REMOVING MUTABLE STORAGE

Now that we can freely retrieve immutable structures, we can focus on other storage variables. A challenge we faced is

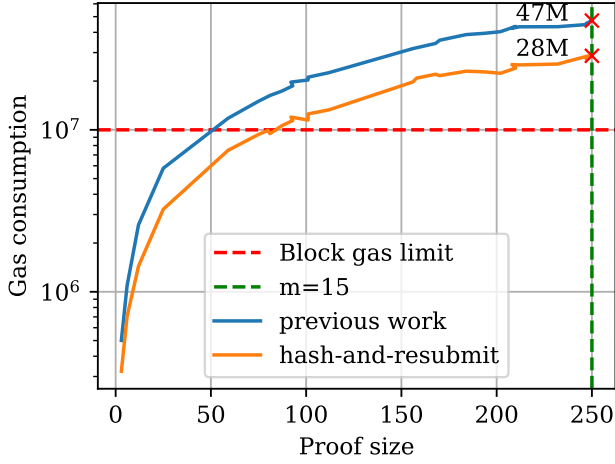


Figure 5: Performance improvement using hash-and-resubmit pattern in NIPoPoWs related to previous work for a secure value of m . The gas consumption decreased by approximately 40%.

that the protocol of NIPoPoWs depends on *DAG* structure which is a mutable hashmap. This logic is intuitive and efficient to implement in most traditional programming languages such as C++, JAVA, Python, JavaScript, etc. However, as our analysis demonstrates, such an 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, 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 construction of an architecture that does not require *DAG*, *ancestors* or any other complementary structures. We will use the notation from [13] 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 [13, 15] that is true for π_{subm} . The case in which b is absent from π_{subm} is aimless, because the submission automatically fails since the predicate is evaluated against π_{subm} . We will refer to such aimless actions as *irrational* and components that are included in such actions as irrational components, i.e. irrational proof, blocks etc. We will use the term *rational* to describe non-irrational actions and components.

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} , then the contest is irrational.

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 $\pi_{subm}\{lca\}$ 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 $b = B$.

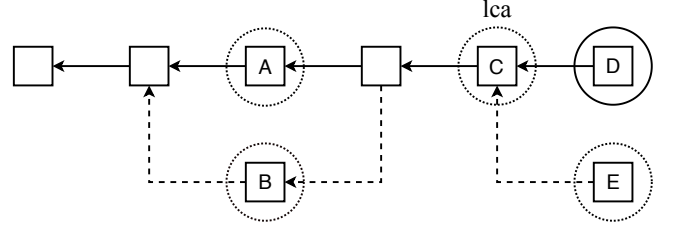


Figure 6: Fork of two chains. Solid lines connection blocks of π_{subm} and dashed lines connect blocks of π_{cont} . Given the configuration, blocks in dashed circles are irrational blocks of interest, and the block i in the solid circle is a rational block of interest.

In Figure 6 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_{cont}^{tr} ⁴. Security is preserved if we require π_{cont}^{tr} 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_{cont}^{tr} is a minimal fork of π_{subm} , we prevent an adversary from dispatching an augmented π_{cont}^{tr} to claim better score against π_{subm} . Such an attempt is displayed in Figure 7.

In Algorithm ??, we show how minimal fork technique is incorporated in our client, replacing *DAG* and *ancestors*. In Figure 8 we show how the performance of the client has improved.

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

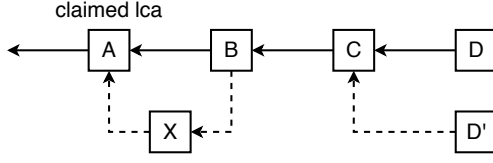


Figure 7: An adversary tries to send a malformed proof consisting of blocks $\{A, X, B, C, D'\}$ that achieves better score against $\{A, B, C, D\}$. This attempt is rejected due to the minimal-fork requirement.

Algorithm 4 The NIPoPoW client using the minimal fork technique

```

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

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

```

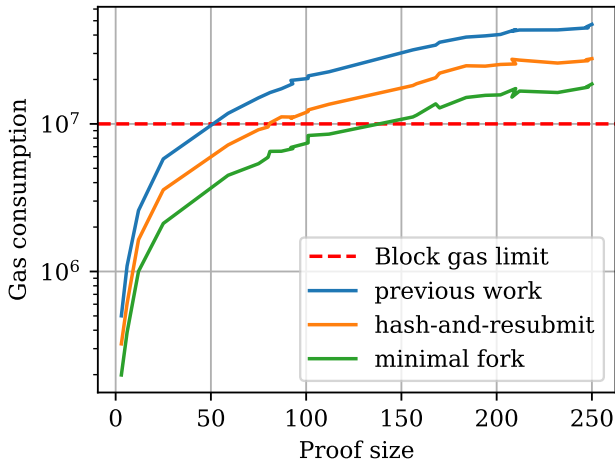


Figure 8

REFERENCES

- [1] Guido Bertoni, Joan Daemen, Michaël Peeters, and GV Assche. 2011. The keccak reference. *Submission to NIST (Round 3)* 13

- (2011), 14–15.
- [2] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. 2020. Flyclient: Super-Light Clients for Cryptocurrencies. (2020).
- [3] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
- [4] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.
- [5] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 81–84.
- [6] Georgios Christoglou. 2018. *Enabling crosschain transactions using NIPoPoWs*. Master's thesis. Imperial College London.
- [7] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
- [8] Kostis Karantias. 2019. *Enabling NIPoPoW Applications on Bitcoin Cash*. Master's thesis. University of Ioannina, Ioannina, Greece.
- [9] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. 2019. Compact Storage of Superblocks for NIPoPoW Applications. In *The 1st International Conference on Mathematical Research for Blockchain Economy*. Springer Nature.
- [10] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. 2019. Proof-of-Burn. In *International Conference on Financial Cryptography and Data Security*.
- [11] Aggelos Kiayias, Peter Gazi, and Dionysis Zindros. 2019. Proof-of-Stake Sidechains. In *IEEE Symposium on Security and Privacy*. IEEE, IEEE.
- [12] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. 2016. Proofs of Proofs of Work with Sublinear Complexity. In *International Conference on Financial Cryptography and Data Security*. Springer, 61–78.
- [13] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. 2020. Non-Interactive Proofs of Proof-of-Work. In *International Conference on Financial Cryptography and Data Security*. Springer.
- [14] Aggelos Kiayias and Dionysis Zindros. 2019. Proof-of-Work Sidechains. In *International Conference on Financial Cryptography and Data Security*. Springer, Springer.
- [15] Yuan Lu, Qiang Tang, and Guiling Wang. 2020. Generic Superlight Client for Permissionless Blockchains. *arXiv preprint arXiv:2003.06552* (2020).
- [16] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. (2009). <http://www.bitcoin.org/bitcoin.pdf>
- [17] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
- [18] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J Knottenbelt. 2019. SoK: Communication across distributed ledgers. (2019).
- [19] Dionysis Zindros. 2020. *Decentralized Blockchain Interoperability*. Ph.D. Dissertation.