

RePoPoW: 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.

KEYWORDS

Blockchain; Superlight clients; NIPoPoWs, Solidity

1 INTRODUCTION

Blockchain interoperability [23] is the ability of distinct blockchains to communicate. This *crosschain* [10, 12, 13, 16, 22] 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) [16]. 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 [18] chain spans roughly 245 GB, and Ethereum [3, 20] has exceeded 350 GB¹. Naturally, not all users are able to accommodate this size of data, especially if portable devices are used, such as mobile phones.

One early solution to compress the extensive size of blockchain is addressed by Nakamoto [18] 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, 11, 14, 15] 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 and dispatched to chain B. Under the condition 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.

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/>

Related Work. We use Non-Interactive Proofs of Proof of Work (NIPoPoWs) [15, 16] 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 [7] provided a Solidity smart contract which is the first implementation of crosschain events verification based on NIPoPoWs, where proofs are submitted and checked for their validity. This solution, however, is impractical due to extensive usage of resources, widely exceeding the Ethereum block gas limit.

Other attempts have been done to address the verification of Bitcoin transactions to the Ethereum blockchain, most notably BTC Relay [6].

Our contribution. Notably, no practical implementation for superlight clients exists to date. In this paper, we focus on constructing a practical client for NIPoPoWs. For the implementation of 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 it is a significant progress in order to provide a widely accepted protocol that enables crosschain transactions. A summary of our contributions in this paper is as follows:

- (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. The security assumptions we make are no others than SPV [9, 21].
- (2) We present a novel pattern which we term *hash-and-resubmit*. Our pattern significantly improves performance of Ethereum smart contracts [3, 20] in terms of gas consumption by utilizing *calldata* space of Ethereum blockchain to eliminate high-cost storage operations.
- (3) We design an *optimistic* schema which we incorporate in the design of our client, that replaces *non-optimistic* operations. This design achieves the improvement of smart contracts' performance by enabling multiple phases of interactions.
- (4) We demonstrate 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, 20]. 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, 8, 19].

By using the *hash-and-resubmit* pattern, large structures are omitted from storage entirely, and are contained in memory. When a function call is performed, the signature and arguments 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 to the contract by other users. To interact publicly with this data without the utilization storage, the node *resends* the observed data to the blockchain. The concept of resending data is redundant in conventional systems. However, this

technique is very efficient to use in Solidity due to the significantly lower gas cost of memory operations in relation with storage operations.

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

The situations in which the *hash-and-resubmit* pattern can be applied are the following:

- (1) The application is a Solidity smart contract.
- (2) Read/write operations are performed in large arrays that exist in storage. Rehashing variables of small size may result to negligible gain or even performance loss.
- (3) The entity that operates on the structures is a full node and observes function calls to the smart contract.

Participants and collaborators. The first participant is the smart contract S that accepts function calls. Another participant is the invoker E_1 , who dispatches a large array d_0 to S via a function $\text{func}_1(d_0)$. Note that d_0 is potentially processed in func_1 , resulting to d . The last participant is the observer E_2 , who is a full node that observes transactions towards S in the blockchain. This is possible because nodes maintain the blockchain locally. After observation, E_2 retrieves data d . Since this is an off-chain operation, a malicious E_2 potentially alters d before interacting with S . We denote the potentially modified d as d^* . Finally, E_2 acts as an invoker by making a new call to S , $\text{func}_2(d^*)$. 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 func_2 .

Implementation. The implementation of this pattern is divided in two parts. The first part covers how d^* is retrieved by E_2 , whereas in the second part the verification of $d = d^*$ is realized. The challenge here is twofold:

- (1) Availability: E_2 must be able to retrieve d without the need of accessing on-chain data.
- (2) Consistency: E_2 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, E_1 makes the function call $\text{func}_1(d_0)$. This transaction, which includes a function signature (func_1) and the corresponding data (d_0), is added in a block by a miner. Due to blockchain's transparency, the observer of func_1 , E_2 , retrieves a copy of d_0 , without the need of accessing contract data. In turn, E_2 performs *locally* the same set of on-chain instructions operated on d_0 generating d . Thus, availability is addressed through observability.

Addressing consistency We prevent an adversary E_2 from altering d^* by storing the *signature* of d in contract's state during the execution of $\text{func}_1(d)$ by E_1 . In the context of Solidity, a signature of a structure is the digest of the structure'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 simplistic 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) the submit phase and (b) the 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 $\text{compare}(a_2, a_1)$ is true, then the holder of the contract changes to P_2 . We provide a simple implementation for $\text{compare}()$, but we can consider any notion of comparison, 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. In Algorithm 1 we display the storage implementation, while in Algorithm 2 we show the implementation embedding the *hash-and-resubmit* pattern.

Algorithm 1 best array using storage

```

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

```

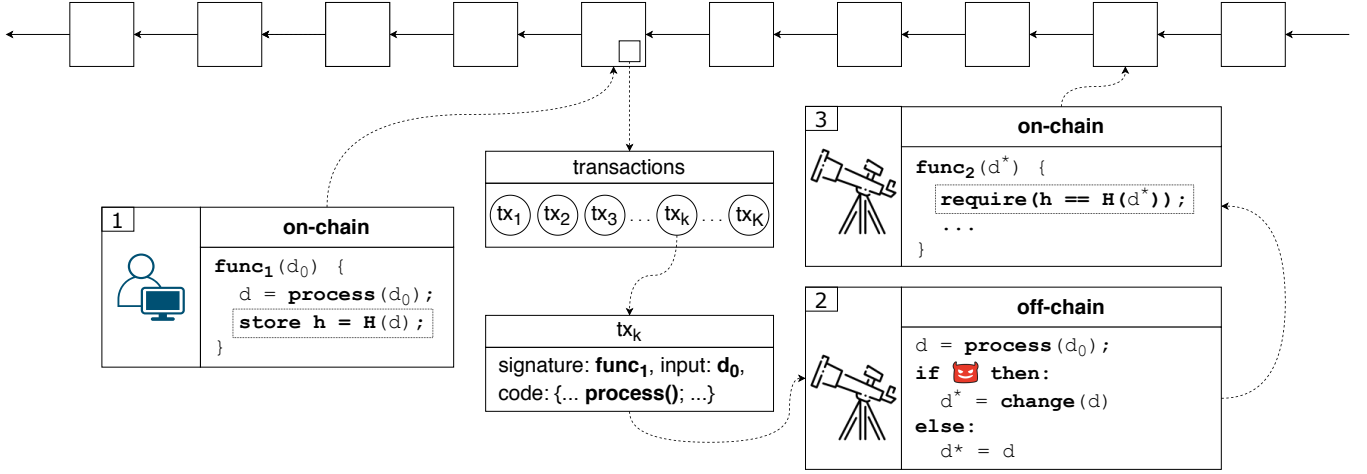


Figure 1: The *hash-and-resubmit* pattern. In Stage 1, an invoker calls `func1(d0)`. `d0` is processed on-chain and `d` is generated. The signature of `d` is stored in the blockchain as the digest of a hash function `H()`. In Stage 2, a full node that observes invocations of `func1` retrieves `d0`, and 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.

Algorithm 2 best array using hash-and-resubmit pattern

```

1: contract best-array
2:   hash  $\leftarrow \emptyset$ ; holder  $\leftarrow \emptyset$ 
3:   function submit(a1)
4:     hash  $\leftarrow H(a_1)$   $\triangleright$  hash saved in storage
5:     holder  $\leftarrow$  msg.sender
6:   end function
7:   function contest(a1*, a2)
8:     require(hash256(a1*) = hash)  $\triangleright$  validate a1*
9:     require(compare(a1*, a2))
10:    holder  $\leftarrow$  msg.sender
11:  end function
12:  function compare(a1*, a2)
13:    require(|a1*|  $\geq$  |a2|)
14:    for i in |a1*| do
15:      require(a1*[i]  $\geq$  a2[i])
16:    end for
17:  end function
18:  return true
19: end contract

```

Gas analysis. The gas consumption of the two above implementations is displayed in Figure 2. By using the *hash-and-resubmit* pattern, the aggregated 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 of size 500 and above, contrary to the optimized version, which consumes approximately only 1/10th of the block gas limit for arrays of 1000 elements.

Variations. In order to enable selective dispatch of a segment of interest, different hashing schemas can be adopted, such as Merkle Trees (ref) and Merkle Mountain Ranges (ref). In this variation of the pattern, which we term *merkle-hash-and-resubmit*, the signature of an array `d` is Merkle Tree Root (MTR). In *resubmit* phase, `d[m:n]` is dispatched, accompanied by the siblings that reconstruct the MTR of `d`.

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 `d`, `|d|` hashes are needed for the leafs of the MT, and `|d| - 1` hashes are needed for the intermediate nodes. For the verification, the segment of interest `d[m:n]` and the siblings of the MT are hashed. The size of siblings is approximately $\log_2(|d|)$. The process of constructing and verifying the MTR is displayed in Figure 3.

In Solidity, different hashing operations vary in cost. An invocation of `sha256(d)`, copies `data` in memory, and then the `CALL` instruction is performed by the EVM that calls a pre-compiled 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 [20]. Consequently, the expression `1 \times sha256(d)` costs less than `|d| \times sha256(1)` operations. A different cost policy applies for `keccak` [1] hash function, where hashing costs 30 gas units plus 6 additional

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

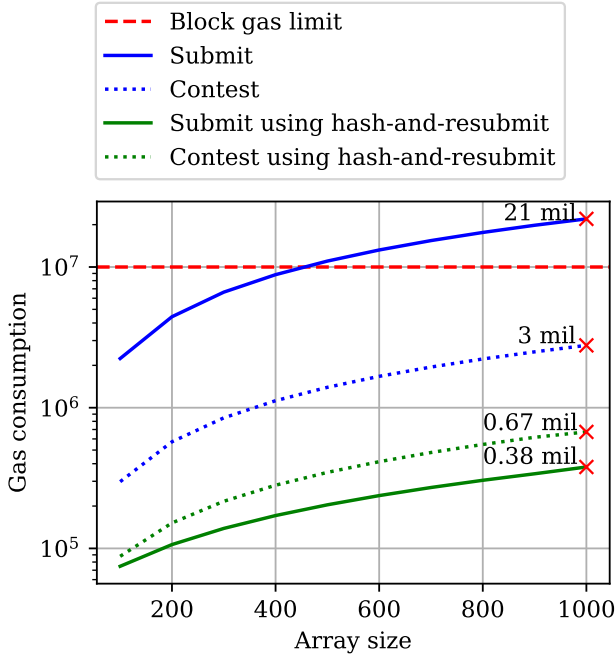


Figure 2: Gas-cost reduction using the *hash-and-resubmit* pattern. By avoiding gas-heavy storage operations, the aggregated cost of submit and contest is decreased significantly by 95%.

Operation	Gas cost
load(d)	$d_{bytes} \times 68$
sha256(d)	$d_{words} \times 3 + 700$
keccak(d)	$d_{words} \times 6 + 30$

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

gas far each word for input data [20]. The usage of keccak dramatically increases the performance in comparison with sha256, and performs better than plain rehashing if the product of on-chain processing is sufficiently larger than the originally dispatched data. Costs of all related operations are listed in Table 1.

The merkle variation can be potentially improved by dividing d in larger chunks than 1 element. We leave this analysis for future work.

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 dispatched data of 10KB, and varying size of on-chain process product.

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

phase per variance	plain hash and resubmit	merkle hash and resubmit
hash	$H(d)$	$H(d_{elem}) \times d $ $H(digest) \times (d - 1)$
resubmit	$load(d) + H(d)$	$load(d[m:n]) +$ $load(siblings) +$ $H(d[m:n]) +$ $H(digest) \times siblings $

Table 2: Summary of operations for *hash-and-resubmit* pattern variations. d is the product of on-chain operations and d_{elem} is an element of d . H is a hash function, such as sha256 or keccak, $digest$ is the product of $H(.)$ and $siblings$ are the siblings of the Merkle Tree constructed for d .

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 is used in the context of the NIPoPoW superlight client. The NIPoPoW verifier adheres to a submit-and-contest-phase schema, and the inputs of the functions are arrays that are processed on-chain. It is, therefore, a suitable case for our pattern.

In *submit* phase, a *proof* is submitted, which, in the case of falsity, it is contested by another user in *contest* phase. The user that initiates the contest is a node and monitors the traffic of the verifier. 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 $m = 15$ to be sufficiently secure.

In previous work, NIPoPoW proofs are maintained on-chain, resulting to extensive storage operations that limit the applicability of the contract considerably. In our implementation, proofs are not stored on-chain, and π_{subm} is retrieved by the node from calldata. Since we assume a trustless network, π_{subm} potentially alters by the node. We denote the potentially changed π_{subm} as π_{subm}^* . In *contest* phase, π_{subm}^* and π_{cont} are dispatched in order to enable the *hash-and-resubmit* pattern.

For our analysis, we create a chain similar to the Bitcoin chain with the addition of the interlink structure in each block as in [7]. Our chain spans 650,000 blocks, which represent a slightly enhanced Bitcoin chain⁴. From the tip of our chain, we branch two chains that span 100 and 200 additional

⁴Bitcoin spans 631,056 blocks as on May 2020. Metrics by <https://www.blockchain.com/>

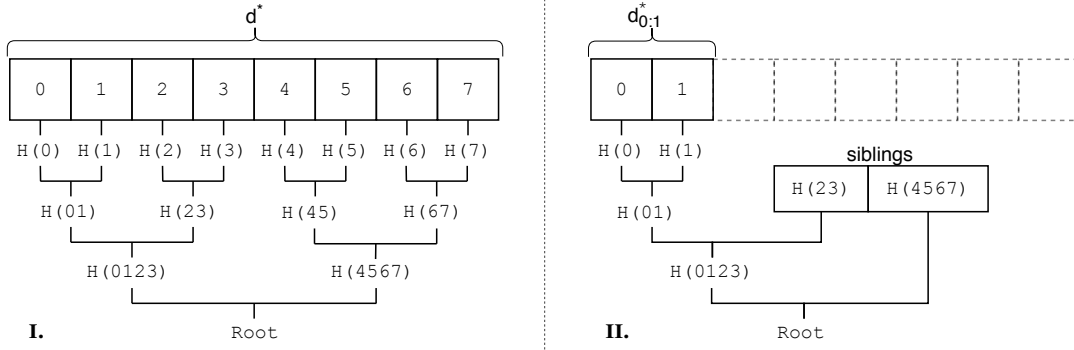


Figure 3: I. The calculation of root in *hash* phase. II. The verification of the root in *resubmit* phase. $H(k)$ denotes the digest of element k . $H(kl)$ denotes the result of $H(H(k) \parallel H(l))$

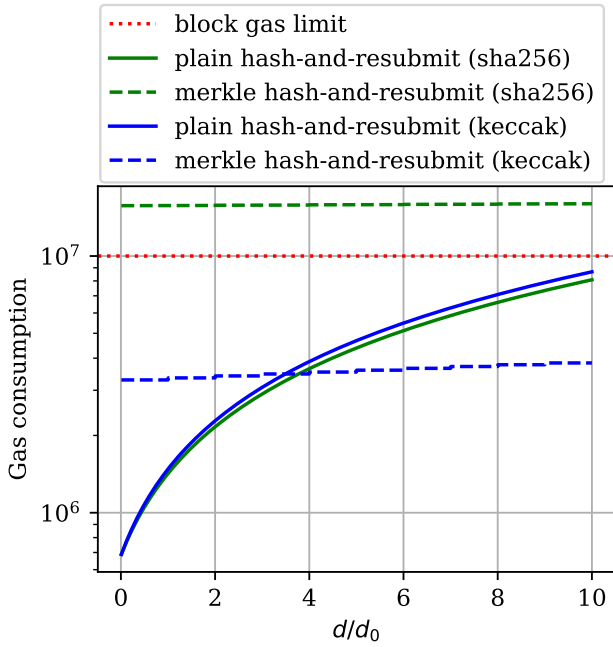


Figure 4: Trade-offs between *hash-and-resubmit* variations. In the vertical axis the gas consumption is displayed, and in vertical axis the size of d as a function of d_0 . The size of d_0 is 10KB bytes, and the hash function we used is pre-compiled sha256.

blocks, respectively, as illustrated in Figure 5. Then, we use the smaller chain to create π_{subm} , and the larger chain to create π_{cont} . By applying the protocol, π_{subm} is submitted, and a contest is initiated with π_{cont} . The contest is successful, since π_{subm} represents a chain consisting of fewer blocks than π_{cont} , therefore encapsulating less proof of work. We select this setting as it provides maximum code coverage, and it describes the most gas-heavy scenario.

In Algorithm 3 we show how *hash-and-resubmit* pattern is embedded into the NIPoPoW client.

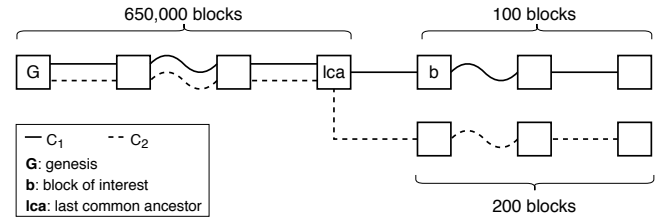


Figure 5: Forked chains for our gas analysis.

In Figure 6, 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 50%. This is a decisive step towards creating a practical superlight client.

4 REMOVING LOOK-UP STRUCTURES

Now that we can freely retrieve large array structures, we can focus on other types of storage variables. A challenge we faced is that the protocol of NIPoPoWs depends on a Directed Acyclic Graph (DAG) of blocks which is a mutable hashmap. This DAG is needed because interlink of superblocks can be adversarially defined. Using it, simple graph search extract the set of ancestor blocks of a block. In the final predicate evaluation, the set of ancestors of the best blockchain tip is passed to the predicate. The ancestors are included to avoid an adversary who presents an honest chain but skips the blocks of interest.

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 implementation in Solidity is significantly

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

```

1: contract crosschain
2:   events  $\leftarrow \perp$ ;  $\mathcal{G} \leftarrow \perp$ 
3:   function initialize( $\mathcal{G}_{remote}$ )
4:      $\mathcal{G} \leftarrow \mathcal{G}_{remote}$ 
5:   end function
6:   function submit( $\pi_{subm}, e$ )
7:     require( $\pi_{subm}[0] = \mathcal{G}$ )
8:     require(events[e] =  $\perp$ )
9:     require(valid-interlinks( $\pi$ ))
10:    DAG  $\leftarrow$  DAG  $\cup \pi_{subm}$ 
11:    events[e].hash  $\leftarrow H(\pi_{subm})$   $\triangleright$  enable pattern
12:    ancestors  $\leftarrow$  find-ancestors()
13:    events[e].pred  $\leftarrow$  evaluate-predicate(ancestors, e)
14:    ancestors =  $\perp$ 
15:  end function
16:  function contest( $\pi_{subm}^*, \pi_{cont}, e$ )  $\triangleright$  provide proofs
17:    require(events[e].hash =  $H(\pi_{subm}^*)$ )  $\triangleright$  verify  $\pi_{subm}^*$ 
18:    require( $\pi_{cont}[0] = \mathcal{G}$ )
19:    require(events[e]  $\neq \perp$ )
20:    require(valid-interlinks( $\pi_{cont}$ ))
21:    lca = find-lca( $\pi_{subm}^*, \pi_{cont}$ )
22:    require(score( $\pi_{cont}[:lca]$ ) > score( $\pi_{subm}^*[:lca]$ ))
23:    DAG  $\leftarrow$  DAG  $\cup \pi_{cont}$ 
24:    ancestors  $\leftarrow$  find-ancestors(DAG)
25:    events[e].pred  $\leftarrow$  evaluate-predicate(ancestors, e)
26:    ancestors =  $\perp$ 
27:  end function
28: end contract

```

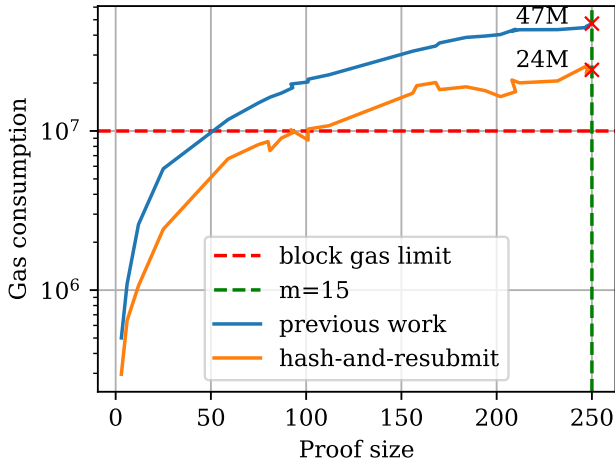


Figure 6: Performance improvement using hash-and-resubmit pattern in NIPoPoWs compared to previous work for a secure value of m . The gas consumption decreased by approximately 50%.

expensive. Albeit Solidity supports hashmap look-up structures, hashmaps can only be contained in storage. This affects performance, especially for large proofs.

We make a keen observation regarding potential positions of the *block of interest* in the proof which lead us to the construction of an architecture that does not require DAG, ancestors or other complementary structures. We adopt the notation from [15] to demonstrate our claims. Additionally, we denote the initially submitted proof as π_{subm} and the contesting proof as π_{cont} . In this context, we consider the predicate e to be of type "block b exists inside a proof", where b denotes the block of interest. The entity that initiates a submission is E_{subm} , and the entity that initiates a contest is E_{cont} .

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 lies at a certain index within π_{subm} and indicates a stable predicate [15, 17] that is true for π_{subm} . A submission in which b is absent from π_{subm} is aimless, because it automatically fails since no element of π_{subm} satisfies e . On the contrary, π_{cont} , tries to prove the falseness of the underlying predicate. This means that, if the block of interest is included in π_{cont} , then the contest is aimless. We 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 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 $\pi_{cont}\{ :lca \}$ can be omitted, because no block B exists such that $\{B : B \notin \pi_{subm}\{ :lca \} \wedge B \in \pi_{cont}\{ :lca \}\}$ where B results into positive evaluation of the predicate. This is due to the fact that, in a rational contest, b is not included in π_{cont} . Consequently, π_{cont} is only rational if it forks π_{subm} in a block is prior to b .

In Figure 7 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 because the lca comes after b . Therefore A is an irrational block as a component of an irrational contest. Given this configuration, the only rational block of interest is D and its predecessors (which we do not display in the figure).

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}^{mf} ⁵. Security is preserved if we require π_{cont}^{mf} to be a *minimal fork* of π_{subm} . A minimal fork is a fork

⁵We cannot proceed to further truncation of π_{cont}^{mf} , because in the NIPoPoW protocol blocks within segment $\pi\{ :lca \}$ of each proof are required for the score calculation.

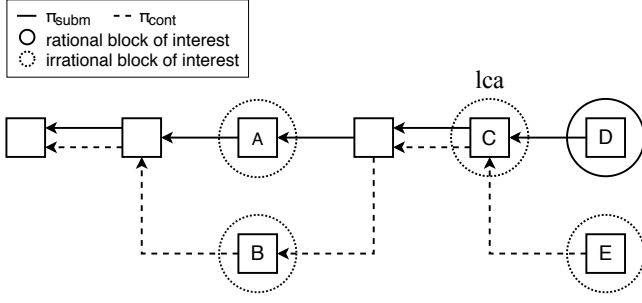


Figure 7: Fork of two chains. Solid lines connect blocks of π_{subm} and dashed lines connect blocks of π_{cont} . In this configuration, blocks in dashed circles are irrational blocks of interest, and the block in the solid circle is a rational block of interest. Blocks B, C and E are irrational because they exist in π_{cont} . Block A is irrational because it belongs to the sub-chain $\pi_{\text{subm}}\{:\text{lca}\}$

chain that shares exactly one common block with the main chain. A proof $\tilde{\pi}$, which is minimal fork of π , has the following attributes:

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

By requiring $\pi_{\text{cont}}^{\text{mf}}$ to be a minimal fork of π_{subm} , we prevent adversaries from dispatching an augmented $\pi_{\text{cont}}^{\text{mf}}$ to claim better score against π_{subm} . Such an attempt is displayed in Figure 8.

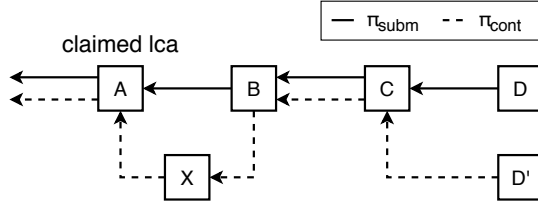


Figure 8: An adversary attests to contest with a malformed proof. Adversary proof consists of blocks $\{A, X, B, C, D'\}$ that achieves better score against submit proof $\{A, B, C, D\}$. This attempt is rejected due to the minimal-fork requirement.

In Algorithm 4, we show how minimal fork technique is incorporated into our client replacing the DAG and ancestors. In Figure 9 we show how the performance of the client improves. We use the same test case as in *hash-and-resubmit*.

TODO: comment results

5 LEVERAGING AN OPTIMISTIC SCHEMA

We discussed how the verification in the NIPoPoW protocol is realized in two phases. In *submit* phase, the verification

Algorithm 4 The NIPoPoW client using the minimal fork technique

```

1: contract crosschain
2:   events  $\leftarrow \perp$ ;  $\mathcal{G} \leftarrow \perp$ 
3:   function initialize( $\mathcal{G}_{\text{remote}}$ )
4:      $\mathcal{G} \leftarrow \mathcal{G}_{\text{remote}}$ 
5:   end function
6:   function submit( $\pi_{\text{subm}}, e$ )
7:     require( $\pi_{\text{subm}}[0] = \mathcal{G}$ )
8:     require(events[e] =  $\perp$ )
9:     require(valid-interlinks( $\pi_{\text{subm}}$ ))
10:    events[e].hash  $\leftarrow H(\pi_{\text{subm}})$ 
11:    events[e].pred  $\leftarrow \text{evaluate-predicate}(\pi_{\text{subm}}, e)$ 
12:  end function
13:  function contest( $\pi_{\text{subm}}^*, \pi_{\text{cont}}^{\text{mf}}, e, f$ )  $\triangleright f$ : fork index
14:    require( $\pi_{\text{cont}}^{\text{mf}}[0] = \pi_{\text{subm}}^*[f]$ )  $\triangleright$  check fork head
15:    require(events[e]  $\neq \perp$ )
16:    require(events[e].hash =  $H(\pi_{\text{subm}}^*)$ )
17:    require(valid-interlinks( $\pi_{\text{cont}}^{\text{mf}}$ ))
18:    require(minimal-fork( $\pi_{\text{subm}}^*, \pi_{\text{cont}}^{\text{mf}}, f$ ))
19:    require(score( $\pi_{\text{cont}}^{\text{mf}}$ ) > score( $\pi_{\text{subm}}^*[f:]$ ))
20:    events[e].pred  $\leftarrow \text{evaluate-predicate}(\pi_{\text{cont}}^{\text{mf}}, e)$ 
21:  end function
22:  function minimal-fork( $\pi_1, \pi_2, f$ )
23:    for  $p$  in  $\pi_1$  do
24:      if  $p \in \pi_2$  then
25:        return false
26:      end if
27:    end for
28:  end function
29: end contract

```

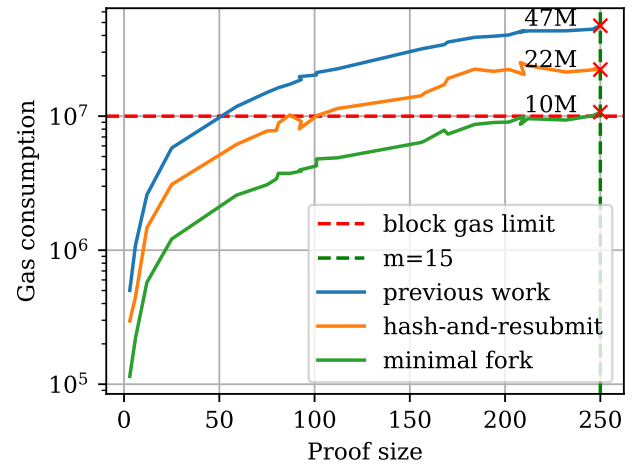


Figure 9

of the π_{subm} is performed. This is necessary in order to prevent adversaries from injecting blocks that do not belong to the chain, or changing existing blocks. A proof is valid for

submission if it is *structurally correct*. Correctly structured NIPoPoWs has the following requirements:

- (1) The first block of the proof is the genesis block of the underlying blockchain.
- (2) Every block has a valid interlink.

Asserting the existence of genesis in the first index of a proof is an inexpensive operation of constant complexity. However, confirming the interlink correctness of all blocks is a process of linear complexity to the size of the proof. Albeit the verification is performed in memory, sufficiently large proofs result into costly submissions. This consists the most demanding function of *submit* phase. In Table 3 we display the cost of *valid-interlink* function which determines the structural correctness of a proof in comparison with the overall gas used in *submit*.

Process	Gas cost	Total %
verify-interlink	2,200,000	53%
submit	4,700,000	100%

Table 3: Gas usage of function *verify-interlink* compared to overall gas consumption of *submit*.

Dispute phase. We observe that the addition of a phase in our protocol alleviates the burden of verifying all elements of the proof by enabling the indication of an individual incorrect block. This phase, which we term *dispute* phase, leverages selective verification of the submitted proof at a certain index, which, as a constant operation, significantly reduces the gas cost of the verification process.

In the protocol where *dispute* is incorporated, when an invalid proof π_{subm} is submitted by E_{subm} , and retrieved by a node E_{cont} from the calldata. Then, the proof is checked for its validity *off-chain*. In order to prove that π_{subm} is invalid, E_{cont} only needs to indicate the index in which π_{subm} fails the interlink verification. Subsequently, E_{cont} calls *dispute*(π_{subm}^*, i), where i indicates the disputing index of π_{subm}^* . Therefore, only one block is interpreted *on-chain*, contrary to the entire span of π_{subm}^* .

Note that, this additional phase does not imply increased rounds of interactions between E_{subm} and E_{cont} . In the case where π_{subm} is invalidated by *dispute* phase, *contest* phase is skipped. Similarly, in the case in which π_{subm} is structurally correct, but represents a chain that is not honest, then E_{cont} proceeds directly to *contest* phase.

In Table 4 we display the gas consumption for two independent cycles of interactions:

- (1) Phases *submit* + *dispute* for a case where π_{subm} is structurally incorrect,
- (2) Phases *submit* + *contest* for a case where π_{subm} is structurally correct, but represents a dishonest chain.

In Algorithm 5, we show the implementation of *dispute* phase in *submit* and *valid-single-interlink*.

Processing fewer blocks. As discussed, *dispute* and *contest* phases are mutually explosive. Unfortunately, constant-time verification cannot be applied in a contest without

	Phase	Gas		Phase	Gas		Phase	Gas
	submit	4.7		submit	2.2		submit	2.2
	contest	4.9		dispute	1.3		contest	4.9
I.	Total	9.6	II.	Total	3.5		Total	7.1

Table 4: Performance per phase. Gas units are displayed in millions. I: Gas consumption prior to dispute phase incorporation. II: Gas consumption for two independent sets of interactions (submit+dispute), (submit+contest).

increasing the rounds of interactions for the users. However, we derive a different major gas optimization for *contest* phase by observing the process of score evaluation.

In NIPoPoWs, after the last common ancestor is found, each fork of the proofs is evaluated in terms of score of proof of work. Each level encapsulates different amount of proof of work, and the level with the best score is the representative of the underlying proof. Since the common portion of the two proofs has the same score, only the disjoint portions need to be addressed. Consequently, the position of the lca determines the span of the proofs that will be included in the score evaluation process. Furthermore, it is impossible to determine the score of a proof in *contest* phase.

On the contrary, when π_{subm} is known, the score of both proofs can be calculated. This means that, after π_{subm} is retrieved from the calldata, the scores of π_{subm} and π_{cont} is known to E_{cont} , as well as the level in which each proof encapsulates the most proof of work. In turn, E_{cont} can only submit the blocks which consist the best level of π_{cont} . The number of these blocks is constant, because it is determined by the security parameter m , which is irrelevant to the underlying blockchain size.

Calculation of best level of π_{cont} is an *off-chain* process. Naturally, an adversarial E_{cont} can intentionally dispatch a level of π_{cont} which is different than the best level. However, this is an irrational act, since different levels only undermine the score of π_{cont} . On the contrary, due to the consistency property of *hash-and-resubmit*, π_{subm} cannot be altered. We denote the best level of $\pi_{\text{cont}}^{\text{mf}}$ as $\pi_{\text{cont}}^{\text{mf,bl}}$.

The utilization of *best-level* methodology greatly increases the performance of the client, because the complexity of the majority of *contest* functions is related to the size of π_{cont} . In Table 5, we demonstrate the difference in *contest* phase after using *best-level*. We observe that the performance of most functions is increased by approximately 85%. This is due to the fact that the size of π_{cont} is decreased accordingly. For $m = 15$, $\pi_{\text{cont}}^{\text{mf,bl}}$ consists of 31 blocks, while $\pi_{\text{cont}}^{\text{mf}}$ consists of 200 blocks. Notably, the calculation of score for $\pi_{\text{cont}}^{\text{mf,bl}}$ needs 0.97% less gas. We observe such a discrepancy because the process of score calculation for multiple levels demands a hashmap which is a storage structure, in contrast with the evaluation of score of a individual level which is done entirely in memory.

In Figure 11, we illustrate the performance gain of the client using *dispute* phase and best-level contesting proof.

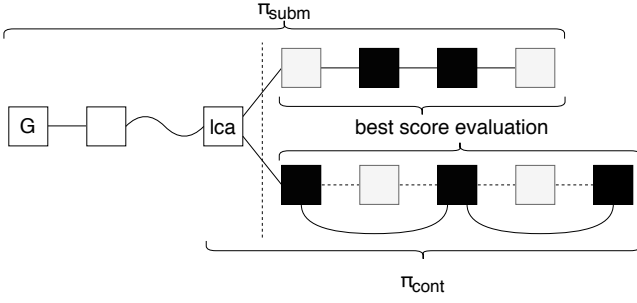


Figure 10: Fork of two proofs. Colored blocks after the lca determine the score of each proof. Black blocks belong to the level that has the best score. Only such blocks are part of the contesting proof.

Process		Gas Cost	Total		Gas Cost	Total
valid-interlinks		900,000	18%		120,000	10%
minimal-fork		1,900,000	39%		275,000	18%
score (p1)		750,000	16%		750,000	51%
score (p2)		950,000	19%		20,000	1%
other		400,000	8%		300,000	20%
contest	I	4,900,000	100%	II	1,465,000	100%

Table 5: Gas usage in contest. I: before utilizing best-level. II: after utilizing best level

The aggregated size of *submit* and *contest* phases is greatly reduced to 3,500,000 gas. This is critical threshold regarding applicability, since a cycle of interactions fit effortlessly inside a single Ethereum block.

6 CRYPTOECONOMICS

Algorithm 5 The NIPoPoW client enhanced with dispute phase and best-level contesting

```

1: contract crosschain
2:   events  $\leftarrow \perp$ ;  $\mathcal{G} \leftarrow \perp$ 
3:   function initialize( $\mathcal{G}_{remote}$ )
4:      $\mathcal{G} \leftarrow \mathcal{G}_{remote}$ 
5:   end function
6:   function submit( $\pi_{subm}, e$ )
7:     require( $\pi_{subm}[0] = \mathcal{G}$ )
8:     require(events[e] =  $\perp$ )
9:     events[e].hash  $\leftarrow H(\pi_{subm})$ 
10:    events[e].pred  $\leftarrow$  evaluate-predicate( $\pi_{subm}, e$ )
11:  end function
12:  function dispute( $\pi_{subm}^*, e, i$ )  $\triangleright i$ : dispute index
13:    require(events[e]  $\neq \perp$ )
14:    require(events[e].hash =  $H(\pi_{subm}^*)$ )
15:    require( $\neg$ valid-single-interlink( $\pi_{subm}, i$ ))
16:    events[e]  $\leftarrow \perp$ 
17:  end function
18:  function valid-single-interlink( $\pi, i$ )
19:     $l \leftarrow \pi[i].level$ 
20:    if  $\pi[i+1].interlink[l] = \pi[i]$  then
21:      return true
22:    end if
23:    return false
24:  end function
25:  function contest( $\pi_{subm}^*, \pi_{cont}^{mf,bl}, e, f$ )
26:    require( $\pi_{cont}^{mf,bl}[0] = \pi_{subm}^*[f]$ )
27:    require(events[e]  $\neq \perp$ )
28:    require(events[e].hash =  $H(\pi_{subm}^*)$ )
29:    require(valid-interlinks( $\pi_{cont}^{mf,bl}$ ))
30:    require(minimal-fork( $\pi_{subm}^*, \pi_{cont}^{mf,bl}, f$ ))
31:    require(score-at-level( $\pi_{cont}^{mf,bl}$ ) > score( $\pi_{subm}^*[f:]$ ))
32:    events[e].pred  $\leftarrow$  evaluate-predicate( $\pi_{cont}^{mf,bl}, e$ )
33:  end function
34:  function score-at-level( $\pi$ )
35:     $l \leftarrow \pi[-1].level$   $\triangleright$  pick proof level from a block
36:     $score \leftarrow 0$   $\triangleright$  set score counter to 0
37:    for b in  $\pi$  do
38:      require(b.level =  $l$ )
39:       $score \leftarrow score + 2^l$ 
40:    end for
41:    return score
42:  end function
43: end contract

```

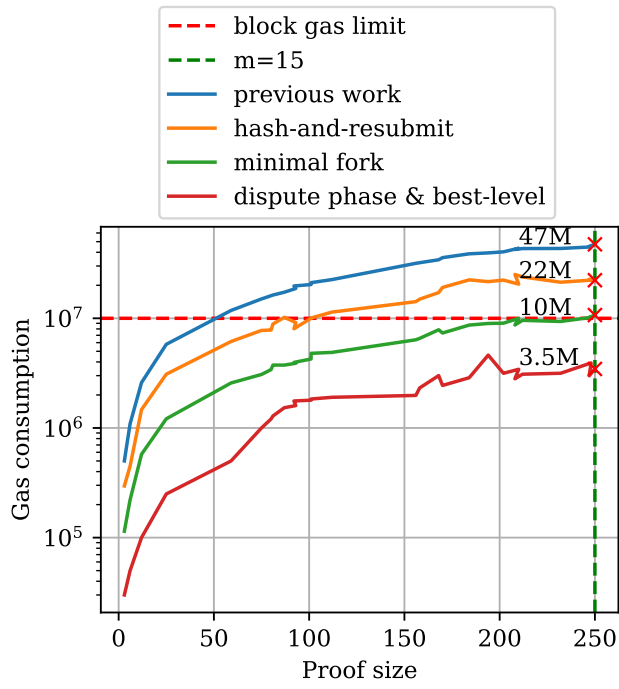


Figure 11: Caption

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] Joseph Chow. 2014. BTC Relay. Available at: <https://github.com/ethereum/btcrelay>. (Dec 2014). <https://github.com/ethereum/btcrelay>
- [7] Georgios Christoglou. 2018. *Enabling crosschain transactions using NIPoPoWs*. Master’s thesis. Imperial College London.
- [8] 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.
- [9] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin’s Peer-to-Peer Network.. In *USENIX Security Symposium*. 129–144.
- [10] Kostis Karantias. 2019. *Enabling NIPoPoW Applications on Bitcoin Cash*. Master’s thesis. University of Ioannina, Ioannina, Greece.
- [11] 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.
- [12] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. 2019. Proof-of-Burn. In *International Conference on Financial Cryptography and Data Security*.
- [13] Aggelos Kiayias, Peter Gazi, and Dionysis Zindros. 2019. Proof-of-Stake Sidechains. In *IEEE Symposium on Security and Privacy*. IEEE, IEEE.
- [14] 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.
- [15] 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.
- [16] Aggelos Kiayias and Dionysis Zindros. 2019. Proof-of-Work Sidechains. In *International Conference on Financial Cryptography and Data Security*. Springer, Springer.
- [17] Yuan Lu, Qiang Tang, and Guiling Wang. 2020. Generic Superlight Client for Permissionless Blockchains. *arXiv preprint arXiv:2003.06552* (2020).
- [18] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. (2009). <http://www.bitcoin.org/bitcoin.pdf>
- [19] Franz Volland. 2018. Memory Array Building. Available at: <https://github.com/fravoll/solidity-patterns>. (April 2018). https://fravoll.github.io/solidity-patterns/memory_array_building.html
- [20] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
- [21] Karl Wüst and Arthur Gervais. 2016. *Ethereum eclipse attacks*. Technical Report. ETH Zurich.
- [22] 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).
- [23] Dionysis Zindros. 2020. *Decentralized Blockchain Interoperability*. Ph.D. Dissertation.