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 [18] is the ability of distinct blockchains to communicate. This *cross-chain* [7, 9, 10, 13, 17] 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) [13]. 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 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 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 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 [15] chain spans roughly 245 GB, and Ethereum [2, 16] 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 [15]. 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 transaction verification, a new generation of *superlight* clients [1, 8, 11, 12] 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 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 the Non-Interactive Proofs of Proof of Work (NIPoPoWs) [12, 13] as the fundamental building block of our solution. This cryptographic primitive is *provably secure* and provides *succinct proofs* regarding the

[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/

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 [5], 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.

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 [2, 16] in terms of gas consumption by utilizing *calldata* space to eliminate high-cost storage operations.
(3) We design an *optimistic* schema for our client which can be used in the design of smart contracts replacing *non-optimistic* architectures. This design achieves the improvement of smart contracts' performance enabling multiple phases of interactions.
(4) 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.

Our implementation meets the following requirements:

(1) Secure: The client is invulnerable against all adversarial attacks.
(2) Trustless: The client is not dependent on third-party applications and operates in a transparent, decentralized manner.
(3) Practical: 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) 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

We now introduce a novel design pattern for Solidity smart contracts which 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 [2, 16]. 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. However, in Ethereum 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 [3, 4, 6].

By using the *hash-and-resubmit* pattern, large storage variables are omitted entirety, and structures are contained in memory which results to 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 available to nodes. By simply observing blocks, a node retrieves data sent by other users, which is processed off-chain. To interact with data originated by other users, the node resends the observed information to the public network, possibly accompanied by complementary data depending on the context of the application. The concept of resending data would be redundant in conventional systems. However, in Solidity this can be utilized very efficiently because function arguments are contained in memory rather than storage, which supports vastly cheaper operations.

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

(1) To reduce gas cost due to 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) When a full node observes the traffic of a contract

Participants and collaborators. The first participant is the smart contract S which accepts function calls. Another participant is the invoker E_1, who dispatches arbitrary data d_0 to S via a function $\mathsf{func}_1(\mathsf{d}_0)$. Note that d_0 are potentially processed on-chain, resulting to d. The last participant is the observer E_2, who is a node that observes transactions towards S in the blockchain. After observation, E_2 retrieves data d. Finally, E_2 acts as an invoker by making a new interaction with S, $\mathsf{func}_2(\mathsf{d})$. However, a malicious E_2 may alter d before

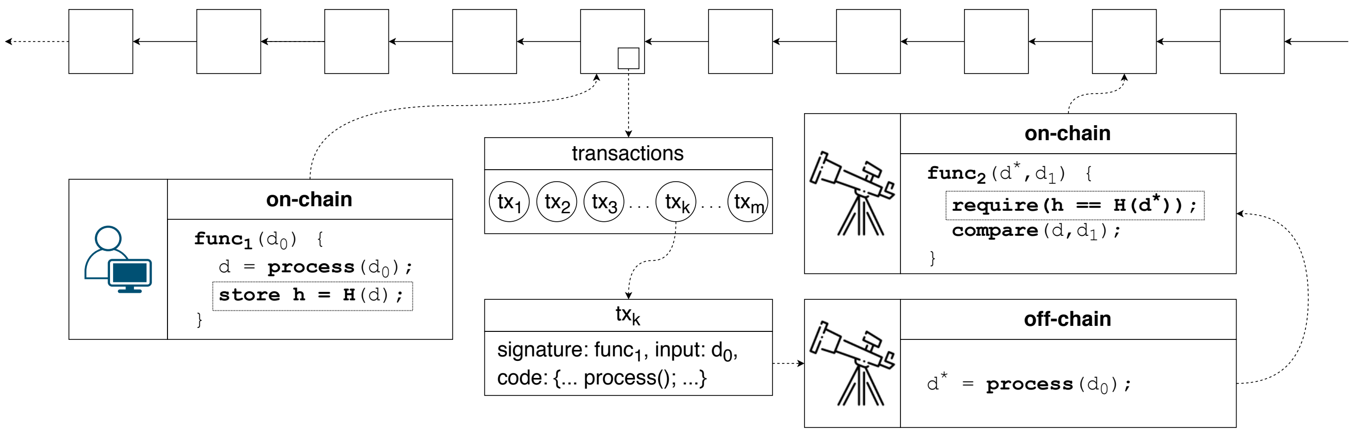

Figure 1: Entity 1 makes a function call with arguments arg1 and arg2. Entity 2 reads the content of transactions in the block and retrieves the input data. Entity 2 can use this data to make a different invocation.

interacting with S. We will denote the potentially modified d as d^*. The verification that $\mathsf{d} = \mathsf{d}^*$, which is a prerequisite for the secure functionality of the underlying contract consists a part of the pattern and is performed in $\mathsf{func}_2(\mathsf{d}^*)$.

Implementation. The implementation of this pattern is divided in two parts. The first part covers how d^* is retrieved by E_2, whereas the second part explains how the verification of $\mathsf{d} = \mathsf{d}^*$ 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 *reliability*, and (b) the *resubmit* phase which addresses *availability* and *reliability*.

Addressing availability: During *hash* phase, E_1 makes the function call $\mathsf{func}_1(\mathsf{d}_0)$. 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.

Addressing reliability: We prevent an adversary E_2 from altering d^* by storing the *hash* of d in contract's state during the execution of $\mathsf{func}_1\texttt{(d)}$ by E_1. The pre-compiled `sha256` is convenient to use in Solidity, however we can make use of any cryptographic hash function $\mathsf{H}()$:

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

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

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

In solidity, 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.

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 and 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 $\mathsf{a}_2 > \mathsf{a}_1$, then the holder of the contract is changed to P_2. We provide a simple implementation for the $>$ operator between arrays, but the comparison can be implemented arbitrarily.

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 implementation using storage, and Algorithm **??** is the implementation embedding the *hash-and-resubmit* pattern.

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 limit[3] for arrays of size 500 and above, contrary to the optimized version, which consumes approximately $1/10^{th}$ of the block gas limit for arrays of 1000 elements.

Merkle-hash-and-resubmit. Now consider a variation of the above game, in which P_1 calls $\mathsf{func}_1\texttt{(}\mathsf{a}_1\texttt{)}$, and then calls $\texttt{pickSpan}(m, n)$ that determines the span of a_1 which can

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

Algorithm 1 best array using storage

```
contract best-array
    function initialize
        best ← ∅; holder ← ∅
    end function
    function submit(a)
        best ← a                          ▷ array saved in storage
        holder ← msg.sender
    end function
    function contest(a)
        require(compare(a))
        holder ← msg.sender
    end function
    function compare(a)
        require(|a| ≥ |best|)
        for i : |best| do
            if a[i] ≤ best[i] then return false
            end if
        end for
        return true
    end function
end contract
```

Algorithm 2 best array using hash-and-resubmit pattern

```
contract best-array
    function initialize
        hash ← ∅; holder ← ∅
    end function
    function submit(a1)
        hash ← H(a1)                      ▷ hash saved in storage
        holder ← msg.sender
    end function
    function contest(a1*, a2)
        require(hash256(a1*) = hash)      ▷ validate a1*
        require(compare(a1*, a2))
        holder ← msg.sender
    end function
    function compare(a1*, a2)
        require(|a1*| ≥ |a2|)
        for i : |a1*| do
            if a1*[i] ≤ a2[i] then return false
            end if
        end for
    end function
    return true
end contract
```

be contested. In reality, P_2 only needs to re-send $\mathsf{a}_1^*[m:n]$ in order to perform the comparison $\mathsf{a}_1[m:n] < \mathsf{a}_2$. However, the digest of a_1 is calculated by hashing the entire structure. Therefore, the *resubmit* phase cannot be successfully performed with $\mathsf{a}_1^*[m:n]$, because $\mathtt{H}(\mathsf{a}_1) \neq \mathtt{H}(\mathsf{a}_1[m:n])$.

An approach to address such scenarios in order to facilitate selective dispatch of structure segments is to adopt different

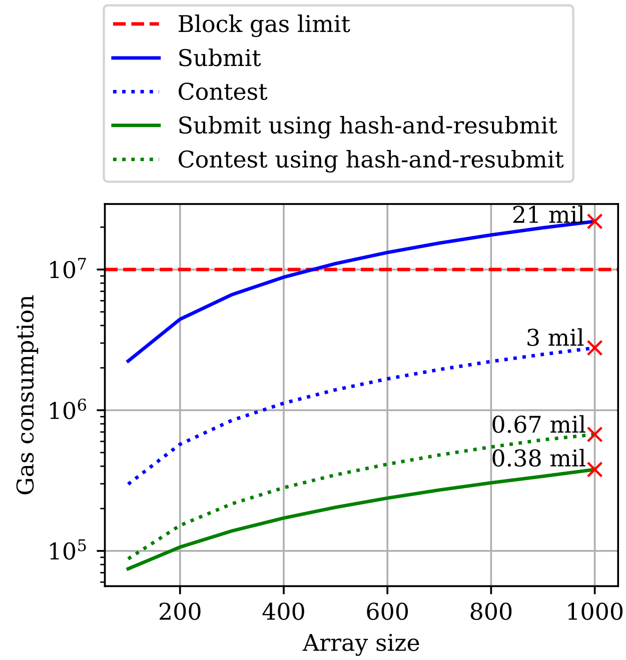

Figure 2: Gas-cost reduction using the *hash-and-resubmit* pattern. By avoiding gas-heavy storage operations, the aggregated cost of submit and contest is decreased significantly by 95%.

hashing schemas that utilize constructions such as Merkle Trees (ref) or Merkle Mountain Ranges (ref). In this variation of the pattern, which we term *merkle-hash-and-resubmit*, the signature of a_1 is generated by constructing the Merkle Root of a_1 in contract's state. In *resubmit* phase, $\mathsf{a}_1[m:n]$ is dispatched, accompanied by the siblings that reconstruct the Merkle Root of a_1 in order to perform the variation of $\mathsf{a}_1^*[m:n]$.

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 Merkle Tree Root for an array a of size n, the hash function $\mathsf{H}(.)$ needs to be called approximately $2^{\lceil log_2(n)+1 \rceil}$ times. For the verification, in order to reconstruct the Merkle Tree Root approximately $log_2 b$ calls to $\mathsf{H}(.)$ are needed. In Solidity, these operations are more expensive than calling $\mathsf{H}(.)$ for the entire span of a once. However, less gas is used to send smaller data structures in *resubmit* phase.

Therefore, the size of the underlying data determines the performance of each variation of the pattern. In Table 1 we display the aggregated gas cost for hashing and verifying the underlying data for both variations as a function of data size. In Figure 3 we demonstrate how gas consumption changes with respect to different sizes of d and $\mathsf{d}_0 = 1$Kb.

actions	hash and resubmit	merkle hash and resubmit
hash	$\mathsf{load}(d_0) + \mathsf{H}(d)$	$\mathsf{load}(d_0) + \mathsf{H}(1) \times 2^{\lceil log_2(\lvert d_0 \rvert+1)\rceil}$
resubmit	$\mathsf{load}(d) + \mathsf{H}(d)$	$\mathsf{load}(d_0^{m:n}) + \mathsf{load}(\lceil log_2(\lvert d_0^{m:n}\rvert)\rceil) + \mathsf{H}(1)\times(\lceil log_2(\lvert d_0^{m:n}\rvert)\rceil)$

Table 1: Summary of operations for *hash-and-resubmit* and *merkle-hash-and-resubmit* patterns. The aggregated gas consumption is determined by the size of the underlying data d_0 and d. d_0 is the data as sent from the caller. d is the product of on-chain operations on d_0, and consist the subject of data comparison between the invoker and the observer of the function. For each variation, different gas cost functions are formulated. In the plain *hash-and-resubmit* pattern, during *hash* phase the hash of d is calculated, which is then verified in *resubmit* phase. 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.

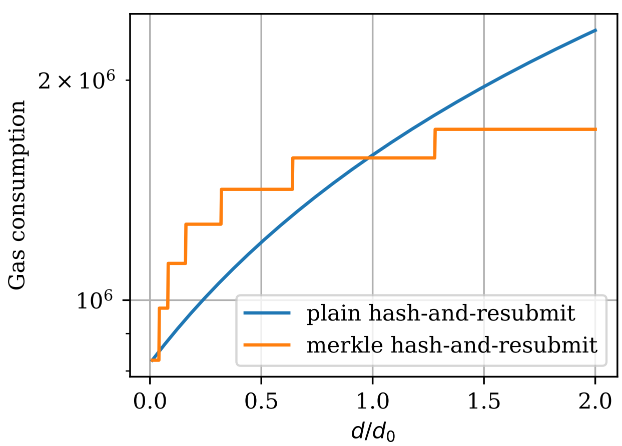

Figure 3: Trade-offs between *hash-and-resubmit* variations. Depending on d_0 and d size, there is an optimal implementation. In vertical axis the gas consumption is displayed, and in vertical axis the size of d as a function of d_0. The size of d_0 is 1000 bytes, and the hash function we used is pre-compiled `sha256`.

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 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 [12]. This is a logical assumption as mentioned in the NIPoPoW paper. 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 process of score evaluation which is described in [12], is irrelevant to the pattern and remains unchained.

In previous work [5], NIPoPoW arrays are saved 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 4, we display how results of the *hash-and-resubmit* implementation differentiate from previous work for the two phases. We observe that by using the *hash-and-resubmit* pattern, we achieve to increase the performance of the contract by reducing the gas consumption by 40%. This is a decisive step towards creating a practical superlight client.

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 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 constriction of an architecture that does not require *DAG*, *ancestors* or any other complementary structures. We will use the notation from [12] 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

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

contract crosschain
  function initialize(G_{remote})
    $\mathsf{G} \leftarrow \mathsf{G}_{remote}$
  end function
  function Submit(π_{subm}, e)
    require($\pi_{\mathsf{subm}}[0] = \mathsf{G}$)
    require($\mathsf{events}[e] = \perp$)
    require($\mathsf{validInterlink}(\pi)$)
    $\mathsf{DAG} \leftarrow \mathsf{DAG} \cup \pi_{\mathsf{subm}}$
    $\mathsf{events}[e].\mathsf{hash} \leftarrow \mathsf{H}(\pi_{\mathsf{subm}})$ ▷ enable pattern
    $\mathsf{ancestors} \leftarrow \mathsf{findAncestors}()$
    $\mathsf{events}[e].\mathsf{pred} \leftarrow \mathsf{evaluatePredicate}(\mathsf{ancestors}, \mathrm{e})$
    $\mathsf{ancestors} = \perp$
  end function
  function Contest(π^*_{subm}, π_{cont}, e) ▷ provide proofs
    require($\mathsf{events}[e].\mathsf{hash} = \mathsf{H}(\pi^*_{\mathsf{subm}})$) ▷ verify π^*_{subm}
    require($\pi_{\mathsf{cont}}[0] = \mathsf{G}$)
    require($\mathsf{events}[e] \neq \perp$)
    require($\mathsf{validInterlink}(\pi_{cont})$)
    $\mathsf{lca} = \mathsf{findLca}(\pi^*_{\mathsf{subm}}, \pi_{\mathsf{cont}})$
    require($\mathsf{score}(\pi_{\mathsf{cont}}[:lca]) > \mathsf{score}(\pi^*_{\mathsf{subm}}[:lca])$)
    $\mathsf{DAG} \leftarrow \mathsf{DAG} \cup \pi_{\mathsf{cont}}$
    $\mathsf{ancestors} \leftarrow \mathsf{findAncestors}(\mathsf{DAG})$
    $\mathsf{events}[e].\mathsf{pred} \leftarrow \mathsf{evaluatePredicate}(\mathsf{ancestors}, \mathrm{e})$
    $\mathsf{ancestors} = \perp$
  end function
end contract

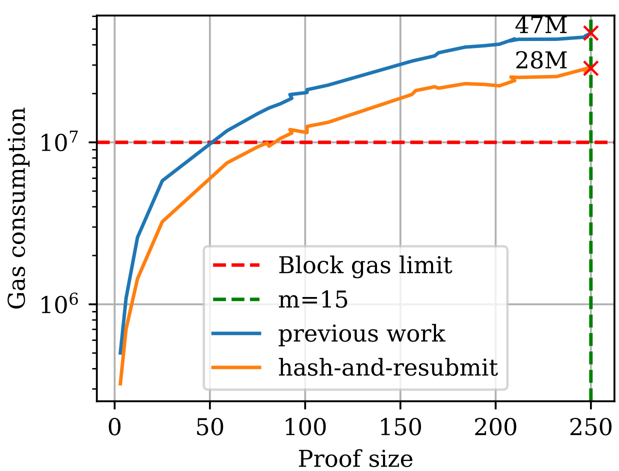

Figure 4: Performance improvement using hash-and-resubmit pattern in NIPoPoWs related to previous work. The gas consumption decreased by approximately 40%

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 [12, 14] 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 predicated is evaluated against π_{subm}. 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. 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 $\pi_{submit}\{:lca\}$ 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 $b = B$.

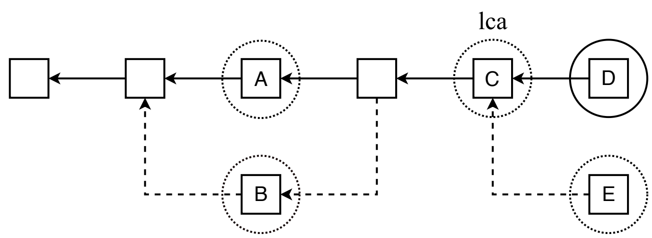

Figure 5: Fork of two chains. Solid lines connection blocks of π_{subm} and dashed lines connect blocks of π_{cont}. Given the configuration, blocks in dashed circles are irrational blocks of interest, and the block i in the solid circle is a rational block of interest.

In Figure 5 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 $\pi^{\mathsf{tr}}_{\mathsf{cont}}$ [4]. Security is preserved if we require $\pi^{\mathsf{tr}}_{\mathsf{cont}}$ 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:

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

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

By requiring that $\pi^{\mathsf{tr}}_{\mathsf{cont}}$ is a minimal fork of π_{subm}, we prevent an adversary from dispatching an augmented $\pi^{\mathsf{tr}}_{\mathsf{cont}}$ to claim better score against π_{subm}. Such an attempt is displayed in Figure 6.

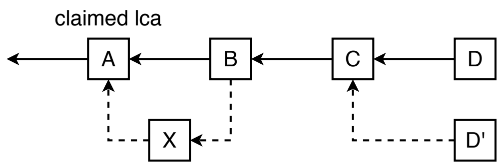

Figure 6: An adversary tries to send a malformed proof consisting of blocks {A, X, B, C, D'} that achieves better score against {A, B, C, D}. This attempt is rejected due to the minimal-fork requirement.

In Algorithm **??**, we show how minimal fork technique is incorporated in our client, replacing *DAG* and *ancestors*. In Figure 7 we show how the performance of the client has improved.

Algorithm 4 The NIPoPoW client using the minimal fork technique

function Submit(π, e)
  require($\pi[0] = Gen$)
  require($events[e]$ = false)
  require($validInterlink(\pi)$)
  $events[e].pred \leftarrow evaluatePredicate(\pi, e)$
  $events[e].hash \leftarrow \mathrm{sha256}(\pi)$
end function

function Contest(π, $\tilde{\pi}$, e, f) ▷ f: fork index in π
  require($\tilde{\pi}[0] = \pi[f]$) ▷ check min. fork head
  require($events[e].hash = \mathrm{sha256}(\pi)$)
  require($events[e].pred$ = true)
  require($validInterlink(\tilde{\pi})$)
  require($disjoint(\pi[f+1:], \tilde{\pi}[1:])$) ▷ check min. fork
  require($score(\tilde{\pi}) > score(\pi[f:])$)
  $events[e].pred \leftarrow evaluatePredicate(\tilde{\pi}, e)$
end function

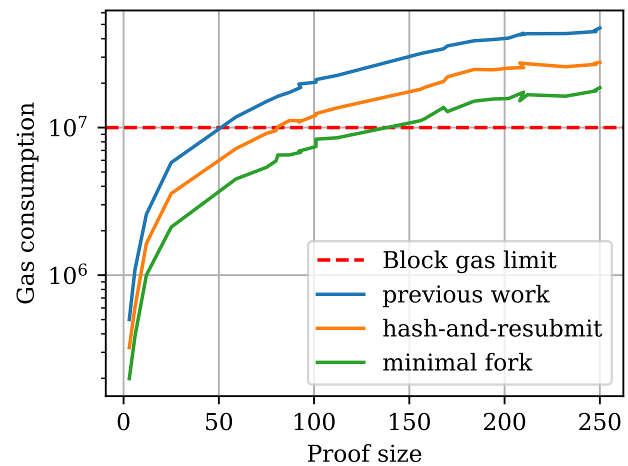

Figure 7

REFERENCES

[1] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. 2020. Flyclient: Super-Light Clients for Cryptocurrencies. (2020).
[2] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
[3] 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.
[4] 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.
[5] Georgios Christoglou. 2018. *Enabling crosschain transactions using NIPoPoWs*. Master's thesis. Imperial College London.
[6] 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.
[7] Kostis Karantias. 2019. *Enabling NIPoPoW Applications on Bitcoin Cash*. Master's thesis. University of Ioannina, Ioannina, Greece.
[8] 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.
[9] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. 2019. Proof-of-Burn. In *International Conference on Financial Cryptography and Data Security*.
[10] Aggelos Kiayias, Peter Gaži, and Dionysis Zindros. 2019. Proof-of-Stake Sidechains. In *IEEE Symposium on Security and Privacy*. IEEE, IEEE.
[11] 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.
[12] 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.
[13] Aggelos Kiayias and Dionysis Zindros. 2019. Proof-of-Work Sidechains. In *International Conference on Financial Cryptography and Data Security*. Springer, Springer.
[14] Yuan Lu, Qiang Tang, and Guiling Wang. 2020. Generic Superlight Client for Permissionless Blockchains. *arXiv preprint arXiv:2003.06552* (2020).
[15] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. (2009). http://www.bitcoin.org/bitcoin.pdf
[16] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
[17] 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).
[18] Dionysis Zindros. 2020. *Decentralized Blockchain Interoperability*. Ph.D. Dissertation.