A Plug-and-Play Long-Range Defense System for Proof-of-Stake Blockchains

Lucien K. L. Ng[1], Panagiotis Chatzigiannis[2], Duc V. Le[3], Mohsen Minaei[4], Ranjit Kumaresan[5], and Mahdi Zamani[6]

[1]`luciengkl@gatech.edu` Georgia Institute of Technology
[2]`pchatzig@visa.edu` Visa Research
[3]`duc.le@visa.edu` Visa Research
[4]`mominaei@visa.edu` Visa Research
[5]`rakumare@visa.edu` Visa Research
[6]`mzamani@visa.edu` Visa Research

Abstract

In recent years, many blockchain systems have progressively transitioned to proof-of-stake (PoS) consensus algorithms. These algorithms are not only more energy efficient than proof-of-work but are also well-studied and widely accepted within the community. However, PoS systems are susceptible to a particularly powerful "long-range" attack, where an adversary can corrupt the validator set retroactively and present forked versions of the blockchain. These versions would still be acceptable to clients, thereby creating the potential for double-spending. Several methods and research efforts have proposed countermeasures against such attacks. Still, they often necessitate modifications to the underlying blockchain, introduce heavy assumptions such as centralized entities, or prove inefficient for securely bootstrapping light clients.

In this work, we propose a method of defending against these attacks with the aid of external servers running our protocol. Our method does not require any soft or hard-forks on the underlying blockchain and operates under reasonable assumptions, specifically the requirement of at least one honest server.

Central to our approach is a new primitive called "Insertable Proof of Sequential Work" (InPoSW). Traditional PoSW ensures that a server performs computational tasks that cannot be parallelized and require a minimum execution time, effectively timestamping the input data. InPoSW additionally allows the prover to "insert" new data into an ongoing InPoSW instance. This primitive can be of independent interest for other timestamp applications. Compared to naïvely adopting prior PoSW schemes for InPoSW, our construction achieves >22× storage reduction on the server side and >17900× communication cost reduction for each verification.

Keywords: Long-Range Attacks, Proof of Sequential Works, Proof-of-Stake

Contents

1 Introduction

Blockchain-based systems provide a high degree of decentralization on distributed trustless ledgers, which have major applications in digital currencies and other financial systems. The original Proof-of-Work (PoW) consensus protocol was part of Bitcoin [26], the first permissionless blockchain, where the parties verifying the validity of the blocks added to the chain (called "miners" or "validators"), had to spend substantial computational power. This was part of a permissionless consensus protocol, with the incentive being monetary rewards. This requirement eventually led to expensive specialized hardware, and very high energy consumption just to reach a consensus, making the system highly inefficient. To resolve these issues, an alternative, more energy-efficient consensus protocol, *Proof-of-Stake* (PoS) was proposed. In a PoS system, validators invest in the blockchain's native tokens, rather than in specialized hardware to participate in the blockchain's validation process. Validators in a PoS system are selected to create new blocks based on their wealth, or *stake*, in the system. Furthermore, to discourage validators from any type of malicious behavior, the PoS protocols typically include a mechanism where the validator's stake is forfeited in case such behavior is detected.

Long-Range Attacks. Despite their popularity in recent years, PoS blockchains are susceptible to a particularly powerful attack called *long-range attack* [10, 9, 23]. This type of attack considers a malicious actor who controls the private keys of validators in the past but no longer have any stake in the system. For instance, an attacker might acquire these keys from exited validators. These validators no longer participate in the consensus process and therefore, cannot be penalized for malicious behavior. Then, assuming the attacker has collected enough voting power at some point in the past using those keys, they can sign a competing block history, which would still be acceptable to the blockchain verifiers. As verifiers would not be able to distinguish the "real" blockchain history from the "fabricated" one, a fork would be created. This fork provides the possibility for the attacker to "double-spend" the amounts they owned before that fork. This can lead to financial losses for the parties that have transacted with the attacker, for example, those who exchanged tokens for physical goods or services.

Possibility of Long-Range Attacks. For popular PoS blockchains with high market capitalization, launching long-range attacks targeting these blockchains requires compromising a large set of validators with high monetary cost and thus is unlikely. However, it does not exclude the possibility of posterior corruption in the future when the blockchain's total value drops. At that period, the past validators might become more allured to selling their old keys to long-range attackers for higher monetary rewards (instead of hiding their keys for the blockchain's reputation and the safety of their assets on the chain). The attacks will become profitable when the blockchain's value bounces back. For similar reasons, PoS blockchains with less popularity or in their infancy might also be more susceptible to long-range attacks because they might have a smaller set of validators, and their coin values are usually more fluctuated.

Checkpoints. To prevent such attacks with devastating consequences, "checkpoints" are typically used in proof-of-stake blockchains. Checkpoints are essentially block header information published periodically on public bulletin boards, block explorers, or other websites. They serve as a reference for validators to discourage such attacks. For instance, Ethereum's Finality Gadgets [3, 14] require a vote of at least 2/3 of validators on a checkpoint to validate it. However, the checkpointing solution is still problematic, as newly onboarded clients need to trust those websites publishing the checkpoints. This introduces a significant degree of centralization in the system, which contradicts the decentralization principles of blockchain. It also creates single points of failure that are susceptible to adversarial corruption.

Existing Impossibility Result. With the checkpointing solution being imperfect, Lewis-Pye and Roughgarden [23] proved that it is impossible to prevent long-range attacks in proof-of-stake consensus without relying on time-dependent cryptographic primitives such as Verifiable Delay Functions (VDFs) or ephemeral keys. More formally, such time-dependent cryptographic primitives can be modeled as oracles that either do not provide an output immediately (e.g., VDFs) or change their output at different times (e.g., key-evolving cryptography).

Shortcoming of Key-Rotation. A naïve solution to long-range attacks is key rotation. Using key-evolving cryptographic primitives, the validators can delete their old keys and update their keys to new ones. With the old keys no longer available, an adversary would not be able to compromise the system. However, this solution suffers from two significant drawbacks: First, it requires that the validators adhere to a new protocol (i.e., a hard-fork). Second, it assumes that validators will delete their old keys, while their incentive would be to keep them and sell these to an attacker for profit after exiting the system.

Our Contributions. We propose a framework that provides defenses against long-range attacks with the following advantages:

- *No need for soft or hard-forks.* Our framework can be adopted on deployed blockchain systems to defend against long-range attacks, without requiring changes to the existing consensus protocol.
- *Reasonable assumptions.* Instead of trusting a few centralized servers that broadcast the history of a blockchain, our protocol works as long as one of the servers that the client connects to is honest.
- *Light-client friendly.* A client only needs to receive a short proof of size logarithmic to the length of the blockchain's history and verify it with logarithmic computation complexity. Therefore, it provides light clients such as resource-constrained devices or web browsers a way to detect long-range attacks in PoS blockchains.

As a core component of our system, we propose a new primitive called Insertable Proof-of-Sequential-Work (InPoSW), which is an extension of Proof-of-Sequential-Work (PoSW). It offers asymptotic and concrete efficiency improvements over prior approaches for timestamping data arriving at different times.

Overview. The participating parties in our system are the following: a) a client without any knowledge about the blockchain's history and current state, and b) an arbitrary number of servers, independent of the existing PoS blockchain validators, with the goal to help defend against long-range attacks. The only requirement is that at least one of these servers is honest.

For each epoch in a PoS blockchain, finalized blocks about the blockchain's history are periodically emitted. Upon receiving a finalized block, a server time-stamps it by importing a digest of the block into an auxiliary sequence generated by an InPoSW algorithm. Timestamping is crucial for defending against long-range attacks because, in such an attack, the attacker can produce another "finalized" block for an early epoch, which would be otherwise valid if the client only inspected the validity of the transactions and PoS consensus results. The timestamp service via InPoSW now comes to rescue by helping the client identify the correct block, which will be the older block produced.

For ease of exposition, we assume there are only 2 servers, where one of them is a long-range attacker and the other is honest. When the client wants to obtain (the commitment of) the current state of the blockchain, it requests the latest state of the chain from both servers. In case of an inconsistency, the client runs a bisection game [29] with the conflicting servers to identify the invalid chains. Still, a long-range attacker can "defeat" the bisection game by providing a valid chain, and the client will see two valid yet different chains differing at an epoch. To resolve this issue, the client asks both servers for the timestamps of that epoch and accepts the chain with blocks that have the older timestamp.

In a PoS blockchain augmented with our proposed framework, it is much harder for an attacker to convince clients to follow a chain generated through a long-range attack. To be able to beat the honest server, the attacker has to prepare the timestamp roughly by the time the blockchain emits an authentic checkpoint; otherwise, the attacker cannot produce a timestamp older than the honest server's and fails to convince the client. Without our defense, an attacker was able to launch an attack by collecting enough old keys in the future, hence the name "long-range". Now, our defense confines the attacker's window of opportunity to a very short period just when the authentic checkpoint is emitted.

Consistency with Prior Impossibility Result. Note that our protocol is consistent with Lewis-Pye and Roughgarden's impossibility result [23], which dictates that any PoS blockchain without time-dependent cryptographic primitives is vulnerable to long-range attacks, as we use VDF to construct an InPoSW as a time-dependent external resource.

Agnostic to PoS Selection Schemes. Our framework works with any PoS blockchain that produces finalized checkpoints, and is independent of the underlying PoS block selection schemes, *e.g.*, Ethereum's GHOST [14] and Ouroboros [21], providing a modular design and high flexibility for deployment.

Efficient InPoSW. As the servers need to timestamp checkpoints produced at different times, they should embed each checkpoint into PoSW instances, which is usually implemented by VDFs or graph-labeling (a.k.a. hash-based) PoSW. VDFs only allow "inserting" data at the beginning of evaluation; thus, running a standalone VDF instance for each data is necessary, implying the servers' computation and storage overhead is linear to the number of inserted data. Graph-labeling PoSW, on the other hand, naturally supports inserting new data during its evaluation. Yet, it requires significantly more storage, computation, and communication on both the servers and the users, mostly due to its sampling techniques when producing proofs. Therefore, we propose a new construction that combines VDFs with the principle of skip-list. Compared to running a VDF, our construction achieves overhead which is logarithmic (instead of linear) to the number of inserted data. Meanwhile, we do not require expensive graph-labeling sampling techniques which greatly reduces the overhead on both sides. Other than defending against long-range attacks, InPoSW may be useful for other applications, *e.g.*, providing cost-efficient large-scale timestamping service.

Incentive of Servers. Our method does not have a direct monetary incentive to the servers. However, by strengthening the security of the blockchain, the servers can earn the blockchain a better reputation and protect their assets on the blockchain, indirectly benefiting them. Their incentives are also analogous to those stemming from other volunteer services such as blockchain explorers.

2 Preliminaries

Basic Notations. We use $X = (x_1, \dots, x_m)$ to denote an array, $X[i]$ to denote the ith item in X, *i.e.*, x_i, $X||y$ to denote a new array constructed by X appended by y. We use $[m]$ to denote an array $(1, 2, \dots, m)$, $\emptyset$ to denote an empty array or set, and $(x_i)_{i \in I}$ to denote an array consisting of x_i with i in an index array I. When the context is clear, *e.g.*, when $I = [m]$ is indices of all the blocks in a blockchain of length m, we may omit I in the subscript. When we say *time*, we refer to the real-time (a.k.a. wall-clock time). *Elapsed time* is the period of time since a prover started timestamping.

Definition 1 (Blockchain) *A blockchain BC produced w.r.t. a blockchain scheme Π is an array $BC = ((B_1, \mathsf{st}_1, \mathsf{FP}_1), \dots, (B_m, \mathsf{st}_m, \mathsf{FP}_m))$ where B_i is a block containing transactions, st_i the state after executing all the transactions in $(B_j)_{j \in [i]}$ according to Π's rules, and FP_i is a finality proof for B_i and st_i.*

Ethereum, for example, has a finality gadget that produces checkpoints signed by 2/3 of validators [3, 14]. A signed checkpoint for an epoch is the finality proof for all the blocks in that epoch. Because of Ethereum's security assumptions and the slashing mechanism, no adversary can produce another 2/3-validator-signed checkpoint for the same epoch, at least within a few weeks. Yet, because of long-range posterior key corruption, an attacker might be able to produce another block in the future. We formalize this concept in the following definition.

Definition 2 (T-Uncorrupted-Finality) *A blockchain scheme Π with T-uncorrupted-finality is for any blockchain $BC = ((B_i, \mathsf{st}_i, \mathsf{FP}_i))_{i \in [m]}$ produced w.r.t. Π, Π guarantees that once FP_m is produced, then within time T, no adversary under the security assumption of Π can produce another chain $BC' = ((B'_i, \mathsf{st}'_i, \mathsf{FP}'_i))_{i \in [m]}$ such that $\mathsf{st}'_m \neq \mathsf{st}_m$, but an honest user would accept BC' over BC even seeing both chains.*

This definition captures the security requirement that once a blockchain finalizes a block, then within a relatively short period of time, there should be no other valid fork of the same (or greater) length. Most blockchains fit into this definition (with various parameter T) because finality is their core requirement.

Bootstrap Protocol is needed by clients who either connect to the blockchain for the first time or have been disconnected from it for an extended period of time. During bootstrapping, a client connects to multiple servers to obtain succinct commitments of the latest state of the blockchain. As long as one of the servers is honest, the client should correctly decide which commitment to accept, in case there is a conflicting output from the servers. While existing bootstrapping protocols work well without the presence of a long-range attacker, our goal is to augment those protocols to defend against such attacks as well. We describe the formal definition in Appendix B

Definition 3 *(Augmented Security against Long-Range Attack) A blockchain scheme Π_T with T-uncorrupted finality coupled with a bootstrap protocol* $\mathsf{Bootstrap}_\beta$ *achieves T'-extended-finality, where $T' = T'(T, \beta)$, against long-range posterior corruption attack, if a new user of Π_T running* $\mathsf{Bootstrap}_\beta$ *can be considered joining a new blockchain scheme that attains T'-uncorrupted-finality under the security assumption of Π_T and* $\mathsf{Bootstrap}_\beta$.

This definition captures the idea that, without modifying the blockchain scheme (*i.e.*, no hard-fork), and by having new users execute the bootstrap protocol with the aid of some servers, launching a long-range attack will be significantly harder, as the bootstrap protocol offers an extended period for uncorrupted finality. We denote by β an "amplifying" parameter stating how much additional security it provides. Looking ahead, our methodology provides multiplicative amplification; namely, $T' = T \cdot \beta$. Sections 5.1 and 6 show the theoretic and estimated concrete parameters of our augmented uncorrupted finality.

Bisection Games. Our construction for the bootstrap protocol relies on *bisection games* to exclude invalid commitments of the blockchain. For a blockchain with m blocks, Tas et al. [29] instantiate bisection games with $O(\log m)$ communication rounds and bandwidthbetween the prover and the verifier and with $O(\log m)$ local computations. Yet, as bisection games are not critical to our conceptual contribution, we postpone its formal introduction to Appendix C.

Definition 4 (InPoSW) *An insertable PoSW scheme* $\mathsf{InPoSW} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ *is a tuple of algorithms performing the following:*

- $\mathsf{Gen}(1^\lambda, \Delta_T) \to \mathsf{pp}$*, where Δ_T is the time interval between two insertions. (The rest of the algorithms implicitly take the* pp *as input).*
- $\mathsf{Eval}(st_N, \mathsf{aux}) \to st_{N+1}$ *takes an old state* st_N *as input and generates the new state* st_{N+1}. Eval *also takes an optional input* aux. *If* aux *is not $\perp$ (null),* Eval inserts *the auxiliary data* aux *into the state, and timestamps this data since then. Each evaluation should timestamp that Δ_T unit of time has elapsed.*
- $\mathsf{Verify}(x, k, \mathsf{st}_N) \to 1/0$ *verifies with the aid of P's state* st_N *that x was inserted $k\Delta_T$ ago into a predecessor of* st_N.

The algorithm should satisfy the following properties:

- *Completeness.* If x is inserted to a predecessor of st_N and has been time-stamped for $k \cdot \Delta_T$ time elapsed via Eval, then $\mathsf{Verify}(x, k, \mathsf{st}_N)$ run with a honest prover and a honest verifier outputs 1.
- *Soundness* (or *sequentiality*[1]). With negligible probability, a PPT adversary $\mathcal{A}$ can produce st_N that makes $\mathsf{Verify}(x, k, \mathsf{st}_N)$ output 1 but $\mathcal{A}$ has less than $(1-O(1))\cdot k\cdot\Delta_T$ sequential execution time since the insertion of x.

[1]Soundness guarantees an adversary cannot have significantly less sequential execution time than an honest prover to construct a valid proof.

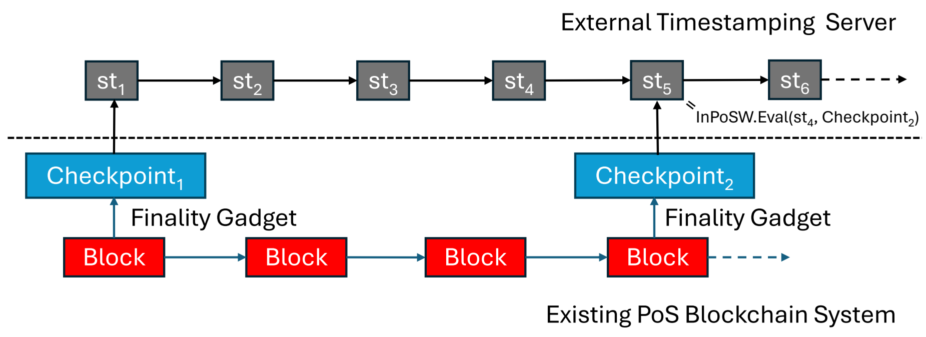


Figure 1: The server's workflow for defending long-range attacks. When the blockchain emits a finalized checkpoint, the server inserts the new checkpoint and the finality proof into its ever-going InPoSW evaluation for timestamping.

- *Succinctness.* The communication and computation cost of Verify imposed to the verifier should be $O(\mathsf{poly}(\lambda, \log \Delta_T, \log N))$

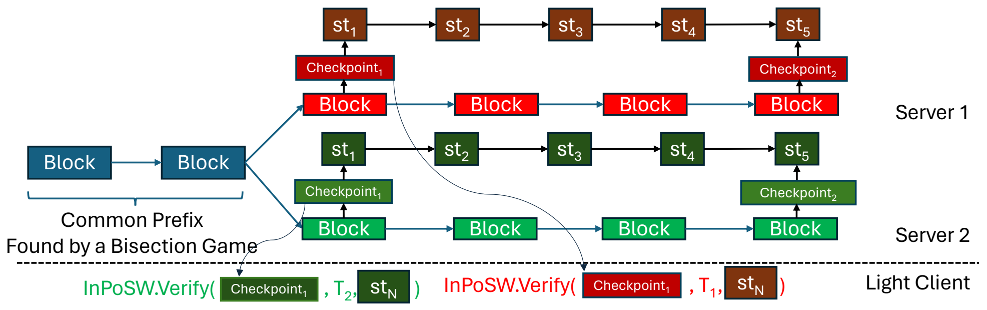
Traditional PoSW [8, 15, 24] can be considered as InPoSW with a large Δ_T and the prover can only run Eval once. InPoSW crucially requires "insertability" on top of PoSW. It allows the prover to timestamp many pieces of data received at different times: after every Δ_T, the prover has an opportunity to "insert" newly arrived data. Additionally, InPoSW supports recurrent invocation of Eval to support ever-going timestamping. Incremental PoSW [18, 6] also supports ever-going evaluation (but still no insertability). Yet, it cannot support soundness and succinctness simultaneously, as we will see in Section 4.1.

Verifiable Delay Function can be considered as a special case of (Traditional) PoSW that additionally requires the output to be unique, unpredictable, and provable. VDFs can be based on various hardness assumptions, *e.g.*, lattice-based schemes [22]. Yet, most practical VDF schemes are based on repeated squaring [12, 30, 27] in unknown order groups, where computing $x^{2^t} \bmod N$ for a (*e.g.*, RSA) modulus N and a known x is believed to be most efficiently computed by t sequential iterative squaring. Wesolowski [30] proposes a practical VDF scheme believed to offer $(1-O(1))\cdot t$-sequentiality, where a PPT adversary can only compute faster than the honest party by a small fraction. More impressively, its output and proof are merely two group elements (the former can be shortened to two integers), and the verification is efficiently done by a few exponentiation and multiplication. Yet, the prover needs to store up to $\sqrt{t}$ group elements during the evaluation. The formal definition is postponed to Appendix B

3 Protocol Description

Timestamping Blockchains. We first describe how a server timestamps a PoS blockchain, without changing the underlying consensus protocol. We assume the blockchain produces a checkpoint for each epoch via a finality gadget, *e.g.*, by letting the last block be signed by the 2/3-majority of the validators Periodically, the blockchain emits a new checkpoint. The server inserts it into the ongoing InPoSW sequence by taking its hash as the auxiliary input to the next state, as shown in Figure 1. Aggregating all checkpoints into a single InPoSW is critical for cost-efficiency because the server only needs to allocate resources to that instance.

Client's Synchronization. When a (light) client wants to synchronize with the blockchain, it connects to multiple servers to acquire the commitment of the most recent blockchain's state. With an honest server and a malicious server, the client will see two different commitments. Similar to many light client protocols [29, 13], the client runs a bisection game with both servers to identify the latest common prefix of both chains and



st1
st2
st3
st4
st5
Checkpoint1
Checkpoint2
Block
Server 1
Server 2
Common Prefix
Found by a Bisection Game
Light Client
InPoSW.Verify(Checkpoint1 , T2, stN)
InPoSW.Verify(Checkpoint1 , T1, stN)

Figure 2: Client's Workflow. In case two servers provide contradicting blockchains, the light client runs a bisection game with the servers to ensure both blockchains are valid (otherwise, accept the only valid one). Then, the servers provide the proofs of InPoSW of their first differing checkpoints. The client accepts the blockchain with a valid proof and more amount of work.

then identifies which chain has the older timestamp via InPoSW.Verify, as shown in Figure 2. The client accepts the chain provided by the server who provides a valid proof with more sequential work.

4 Construction of InPoSW

4.1 Challenges of Constructing InPoSW

Building InPoSW from Graph-Labeling PoSW at first glance seems to be a good blueprint as it can achieve insertability with slight modification. Most graph-labeling PoSW schemes [16, 25, 18, 7, 6] follows the same paradigm: The prover hashes the input into a label, then it repeatedly hashes the last label and (the hashes of) some of the previous labels to compute the latest label, forming a directed acyclic graph where an edge indicates input to the hash, and the nodes are hash labels. For succinct verification, the hashing graph is formed in a way that enables efficient opening of the labels.

During the sequential evaluation of the PoSW graph, one can easily insert new data into the new node in the hashing graph by taking it as additional input for the hash. SNACK [7] formalizes the modification for any PoSW scheme. However, known graph-labeling schemes *cannot* achieve *both* soundness and succinctness, even for SNACK and the state-of-the-art [6]. During verification, these schemes rely on sampling the PoSW hash graph's nodes, and the prover sends to the verifier the openings of these sampled nodes to show their labels' consistency. The number of samples grows *linearly* with the total evaluation time, and therefore, the total proof size.

To show that the prover has done at least $\alpha \in [0, 1]$ fraction of the total number of sequential hashing N with overwhelming probability, *i.e.*, a cheating prover can at most skip $(1 - \alpha) \cdot N$ sequential hashing, the prover needs to provide τ samples of in its proof such that $s \geq \alpha^\tau + \text{other terms}$, requiring $\tau \geq \log s / \log \alpha$. The critical issue is that to prove the skipped evaluation time is bounded by a value Δ_T (*e.g.*, the insertion interval or a small portion of a blockchain's uncorrupted period), α has to grow with N. To illustrate, we denote by δ_H the computation time needed for a work (hashing). If we fix α for all N, the time difference between an honest prover's and a cheating one's work $(1 - \alpha)N\Delta_T$ increases as N increases, allowing the adversary to skip more evaluation time.

To bound an adversary's skipped time by Δ_T, we need to maintain an inequality $(1 - \alpha)\delta_H N \leq \Delta_T$ by adjusting α, requiring $\alpha \geq 1 - \Delta_T/(\delta_H N)$. The needed number of sample τ for N is linear to N:

$$\tau \geq \log s / \log \alpha \geq \log s / \log\,(1 - \frac{\Delta_T}{\delta_H N}) \overset{(*)}{\approx} \log s / \frac{-\Delta_T}{\delta_H N} = N\delta_H \cdot \frac{-\log s}{\Delta_T}$$

Note that $\overset{(*)}{\approx}$ holds because $\log(1 - x) \approx -x$ for $x \approx 0$, and $\Delta_T \ll \delta_H N$ when we want to prove for some meaningful amount of work N. Finally, because the proof size is τ times the size of the opening for each

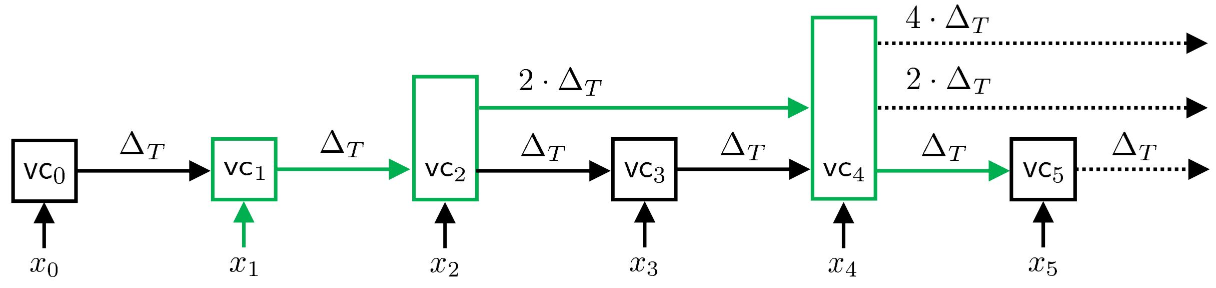


Figure 3: The overview of our InPoSW construction. Each horizontal arrow indicates a VDF evaluation $\mathsf{VDF.Eval}(\mathsf{st}_i, \mathsf{vc}_i)$ with evaluation time proportional to the arrow's length. The solid ones are finished evaluations, while the dotted ones are still ongoing. Each evaluation takes as input a vector commitment vc boxed on the left. Each vc merges all the data that has an arrow pointing to it, including the output of previous $\mathsf{VDF.Eval}(\cdot)$ and the inserted data x_i. To verify, say, x_1 is inserted at time Δ_T, the prover sends all the openings of vcs and VDF proofs on the shortest path from x_1 to the latest finished evaluation, which are in green.

challenge, it is linear to the elapsed time $N\delta_H$, violating the poly-logarithmic complexity requirement for verification.

It might be tempting to reduce verification cost by proving a *constant fraction* of work, instead of proving the prover can at most skip a constant amount of work. In this case, the protocol sticks to a constant α and needs not to increase the number of samples (τ) in the proof. However, such a proof system will allow an adversary to eventually outrun the honest server to forge a proof, regardless of how other components of the system improve, *e.g.*, even if a perfect VDF is invented.

Verifiable Delay Function is helpful for building InPoSW because most VDF schemes do not resort to sampling techniques like graph-labeling PoSW. Thus, the prover does not need to provide linearly larger proofs as the number of evaluations grows. Nevertheless, VDFs alone do not provide *insertability* because the prover has to commit to the data in the beginning and perform sequential computation based on this dataalone, and no new data can be inserted once the evaluation has started. Whenever the prover inserts new data, it needs to start a new standaloneVDF instance. On top of the cost of running each VDF, the prover's storage and computation grows by another factor *linear* to the number of inserted data, while a (basic) VDF can only prove sequentiality for a pre-defined period T. On the other hand, our definition ofInPoSW requires the prover to perform a continuous evaluation and provide proof based on the accumulated evaluations.

4.2 Construction overview

At a high level, our construction relies on the concurrent evaluation of VDFs in a skiplist style, as illustrated in Figure 3. For a prover P that has been running for time $N \cdot \Delta_T$, it will be evaluating $\log_2 N$ VDFs, each of which is set up to prove time linear to Δ_T times $2^0, 2^1, \ldots, 2^{\lceil \log_2 N \rceil}$, respectively. Data can be inserted as part of a new VDF's input when P starts evaluating a new one. To prove to a verifier that data x_t was inserted at time $t \cdot \Delta_T$, P sends the proofs of VDFs evaluated between time $t \cdot \Delta_T$ and now. Because of the skiplist structure, P sends at most $O(\log N)$ proofs by omitting the proofs of VDFs whose evaluation period is entirely overlapped by another VDF's evaluation.

Prover Evaluation. As illustrated in Figure 3, P arranges the VDFs evaluation in a skiplist style. First P starts evaluating a VDF for time $O(\Delta_T \cdot 2^0)$. When this VDF's evaluation is done, P takes this VDF's output (and any inserted data) as input to the next VDF for the same period of time $O(\Delta_T \cdot 2^0)$. Meanwhile, P starts evaluating another VDF with the same input but for time $O(\Delta_T \cdot 2^1)$. In general, when P finishes evaluating a VDF for time $O(\Delta_T \cdot 2^i)$ for some integer i, P starts evaluating another VDF for time $O(\Delta_T \cdot 2^i)$

- Parameters: P decided a bound time resolution Δ_T at Setup
- Input:
    - P: previous state st_N and an auxiliary data aux (which can be null $\perp$)
- Output:
    - P obtains a new state st_{N+1} after evaluation with time Δ_T
- Procedure:
    - `// AUX,` $\mathcal{VC}$`, and` $\mathcal{VDF}$ `are defined as arrays`
    - P retrieves $(N, \mathsf{AUX}, \mathcal{VC}, \mathcal{VDF}) = \mathsf{st}_N$
    - P stores the auxiliary data into $\mathsf{AUX}[N] \leftarrow H(\mathsf{aux})$
    - P gathers the outputs of just finished VDFs for a vector commitment

$$\mathsf{outs} = (\mathcal{VDF}[i][N/2^i].\mathsf{output})_{\{i \in \mathbb{N} \mid 2^i \text{ divides } N \text{ and } 2^i < N\}}$$
$$\mathsf{vc} = \mathsf{VecCom.Commit}(\mathsf{outs}||\mathsf{aux})$$

`//` i`s are the position of '1' bit 1 in` N`.`

- P stores $\mathcal{VC}[N] \leftarrow \mathsf{vc}$ (and implicitly the auxiliary data for openings)
- For each i that 2^i divides N:
    - P prepares $\mathsf{pp} = \mathsf{VDF.Setup}(\lambda_{\mathsf{VDF}}, \Delta_T \cdot 2^i)$ for proving runtime $\Delta_T \cdot 2^i$.
    - P start evaluating $\mathsf{VDF.Eval}(\mathsf{pp}, \mathsf{vc})$ and store this running process into $\mathcal{VDF}[i][N/2^i]$. Once the evaluation is done, P can retrieve the output and proof from this array.
- P stores and outputs $\mathsf{st}_{N+1} = (N+1, \mathsf{AUX}, \mathcal{VC}, \mathcal{VDF})$

- Computation (counting from the beginning):
    - P performs $O(N\Delta_T)$ sequential and $O(N\Delta_T \log(N\Delta_T))$ total computation.
- Storage: P stores $O(N \cdot (\Delta_T \cdot \lambda_{\mathsf{VDF}} \cdot \log N + \mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T))$ bits, storing:
    - Results of $O(N)$ computed VDFs, each with $\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T)$ bits.
    - Commitments and their auxiliary data with $O(\lambda_{\mathsf{hash}} \cdot N \cdot \log N)$ bits.
    - Data for $O(\log N)$ ongoing VDFs, each of which takes $O(\Delta_T \cdot N \cdot \lambda_{\mathsf{VDF}})$ bits.

Figure 4: $\mathsf{InPoSW.Eval}$ where P internally proceeds into the next state for timestamping in our skiplist-style construction and possibly inserts a new data.

and, if no VDF for time $O(\Delta_T \cdot 2^{i+1})$ is evaluating, P also starts evaluating one. The newly started VDFs take as input the vector commitment of the outputs of all VDFs finished at that time. We describe the full protocol in Figure 4.

Verification. Now, we illustrate how P with state st_N can prove to a verifier V that data x is inserted at time $t \cdot \Delta_T$, and that sequential computation has been performing until now $N \cdot \Delta_T$. As illustrated in Figure 5, P retrieves from st_N the VDF proofs on the shortest path from node t to node N. In addition, P needs to prove these proofs are "connected" in a way that on the path, a VDF takes input from its previous VDF's output (except the very first VDF). P can reveal all the inputs to each VDF, which includes the output of its previous VDF. To slightly reduce communication cost, P can replace concatenation by a vector commitment, *e.g.*, a Merkle root, when merging inputs for a VDF. Now, the opening for a VDF's input is $O(\log \log T)$ instead of $O(\log T)$.

Setup. Let P set $\mathsf{pp} = (\lambda, \Delta_T)$ and $\mathsf{st}_0 = (0, \mathsf{AUX} = \emptyset, \mathcal{VC} = \emptyset, \mathcal{VDF} = \emptyset)$.

Theorem 1 *Our construction of* InPoSW *shown in Figure 4 and 5 is complete, sound and succinct.*

We provide a formal proof for the above theorem in the Appendix.

- Parameters: P decided a bound time resolution Δ_T at Setup.
- Input:
  - P and V input k and data x on which they agree to verify that x was inserted $k \cdot \Delta_T$ ago
  - P inputs its current state st_N
- Output:
  - V outputs 0/1 indicating if it accepts P's proof
- Procedure:
  - P retrieves $(N, \mathsf{AUX}, \mathcal{VC}, \mathcal{VDF}) = \mathsf{st}_N$
  - P sends N and $t = N - k$ to V
  - P and V compute $z = 2^{\lfloor \log_2 t \rfloor}$ and a running index $j = N$
  - While $j > t$:
    - P and V find the largest $i \in \mathbb{Z}$ such that
      - if $j > z$, 2^i divides j; `//` i `is the position of the least significant non-zero bit in` j.
      - otherwise, $j - 2^i \geq t$ `//` i `is the position of the most significant non-zero bit in` $j - t$.
    - P retrieves and send to V
      - VDF's pub. param., output, and proof $(\mathsf{pp}, y, \pi) = \mathcal{VDF}[i][j/2^i - 1]$
      - (vector-committed) input $\mathsf{vc} = \mathcal{VC}[j].\mathsf{Com}$
    - V verifies them by computing $b = \mathsf{VDF.Verify}(\mathsf{pp}, \mathsf{vc}, y, \pi)$.
    - If $j \neq N$,
      - P sends to V $\pi_{\mathsf{vc}} = \mathsf{VecCom.Open}(\mathsf{vc}', y, i)$
      - V verifies $b_{\mathsf{vc}} = \mathsf{VecCom.Verify}(\mathsf{vc}', y, i, \pi_{\mathsf{vc}})$ and updates $b = b \wedge b_{\mathsf{vc}}$
    - If $b = 0$, V outputs 0 and aborts.
    - P and V updates $j = j - 2^i$ and marks $\mathsf{vc}' = \mathsf{vc}$
  - P sends to V $\pi_{\mathsf{vc}} = \mathsf{VecCom.Open}(\mathsf{vc}', H(x), \text{last index})$
  - V computes $b = \mathsf{VecCom.Verify}(\mathsf{vc}', H(x), \text{last index}, \pi_{\mathsf{vc}})$
  - V outputs b
- Computation: P and V perform
  - V performs $O(\log N \cdot (\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T + \lambda_H \cdot \log\log N)))$ comp. due to
    - $\leq 2\log_2 N$ VDFs verification, each with $\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T)$ comp.
    - $\leq 2\log_2 N$ VecCom verification, each with $O(\lambda_H \cdot \log\log N)$ comp.
  - P performs $O(\log N \cdot \log\log N \cdot \lambda_H)$ computation due to
    - retrieving $\leq 2\log_2 N$ VDF proofs
    - constructing $\leq 2\log_2 N$ VecCom openings, each with $O(\log\log N \cdot \lambda_H)$ comp.
- communication:
  - P sends to V $O(\log N(\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T) + \lambda_H \cdot \log\log N))$ bits
    - $\leq 2\log_2 N$ VDFs proof tuples, each with $\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T)$ bits.
    - $\leq 2\log_2 N$ commitments and openings, each with $O(\lambda_H \cdot \log\log N)$ bits.

Figure 5: $\mathsf{InPoSW.Verify}$. Note that this protocol is non-interactive because the verifier does not send data to the prover.

5 Construction of Bootstrap against Long-Range Attacks

Our construction $\mathsf{BootBiPoSW} = (\mathsf{Setup}, \mathsf{Prepare}, \mathsf{Verify})$ in Figure 8 (in the Appendix) shows how an honest prover prepares the timestamps running the bootstrap protocol in the future. Setup and $\mathsf{Prepare}$ are solely invoked by a prover when synchronizing with the blockchain and updating its state. Verify is invoked by a verifier V and a group of provers to help the verifier get a vector commitment of the authentic and updated blockchain.

PoSW Scheme	Prover Storage	Proof Size
GL-PoSW (SNACKs [7])	$O(N \cdot \Delta_T \cdot \lambda_H)$	$O(N \cdot \log^2(N \cdot \Delta_T) \cdot \lambda_H)$
Our InPoSW w/ Blackbox VDF	$O(N \cdot (\Delta_T \cdot \lambda_{\mathsf{VDF}} \cdot \log N + \mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T))$	$O(\log N \cdot (\mathsf{poly}(\lambda_{\mathsf{VDF}}, \log N\Delta_T) + \lambda_H \cdot \log\log N))$
Our InPoSW w/ Efficient VDF [30]	$O(\log N \cdot (N \cdot (\lambda_{\mathsf{VDF}} + \lambda_H) + \sqrt{N \cdot \Delta_T} \cdot \lambda_{\mathsf{VDF}}))$	$O(\log N \cdot (\lambda_{\mathsf{VDF}} + \lambda_H \cdot \log\log N))$

Table 1: Asymptotic Cost of Insertable PoSW. Verification computation complexity is roughly similar to the proof size complexity. N is the number of evaluations. Δ_T is the time interval between insertion λ_H and λ_{VDF} are the security parameters of the hash function and VDF, respectively.

Cost Analysis. The worst-case cost of our bootstrap protocol is the sum of the cost of the bisection game and the insertable PoSW. Our bootstrap protocol is succinct as both sub-protocols are succinct, meaning that the overall verification cost is poly-logarithmic to the number of blocks (and elapsed time) and polynomial to the security parameters. As our insertable PoSW is non-interactive, the bootstrap protocol can be made non-interactive if the bisection game (for the blockchain scheme)is also non-interactive.

Server Maintenance. Our construction also allows easy maintenance. When the server wants to shut down old hardware and upgrade to new ones, it can wait until a time slot whose index is a power-of-2, *i.e.*, the time is $\Delta_T \cdot 2^i$. In this case, the server only needs to transfer a (succinct) vector commitment to the new equipment, minimizing the downtime due to data transfer. The proofs of prior VDF evaluation can be transferred later as they do not affect the ongoing evaluation. On the other hand, due to its sampling technique, graph-labeling PoSW requires the server to transfer the entire graph to the new hardware, which might be equivalent to GB or TB of data and lead to disruption.

5.1 Security

Theorem 2 *For a blockchain scheme Π with T_Π-uncorrupted-finality, by coupling with the bootstrap protocol* $\mathsf{BootBiPoSW}$ *using VDF with $(1-\epsilon)\cdot T$-sequentiality, Π achieves $T_\Pi \cdot (1/\epsilon)$-extended-finality against long-range attacks.*

We provide a formal proof for the above theorem in the Appendix. Looking ahead, our scheme guarantees that if there exists a VDF where adversaries can at best evaluate the VDF, for example, 1% faster, we can augment the security of a T-uncorrupted-finality blockchain to achieve $100 \cdot T$-augmented-security.

6 Performance Estimation with Concrete Parameters

Here, we estimate the storage and communication costs of insertable PoSW schemes which will dictate the operational costs imposed on the server (the prover) and the verification costs of the client (the verifier). Our estimation shows that verification with our InPoSW construction with a properly chosen VDF scheme requires only sending KB-size proofs, *i.e.*, light-client friendly.

Asymptotic Comparison. Table 1 shows the asymptotic prover storage cost and proof size. In contrast to SNACKs' proof size which is linear to the number of evaluations N, our InPoSW's proof size is logarithmic to N, achieving succinctness. We show the asymptotic cost of our InPoSW by plugging a constant-proof-size VDF scheme [30]. With this VDF scheme, storing a tuple of VDF proof and output merely costs $\lambda_{\mathsf{VDF}} + 2\lambda_H$ bits. During each VDF's evaluation, the prover needs to memorize $\approx \sqrt{N \cdot \Delta_T} \cdot \lambda_{\mathsf{VDF}}$ bits.

Computation complexity for Verification is roughly similar to the proof size. For SNACKs, the verifier needs to compute $O(N \cdot \log^2 N)$ hashes. The most expensive computation of our adopted VDF [30] is

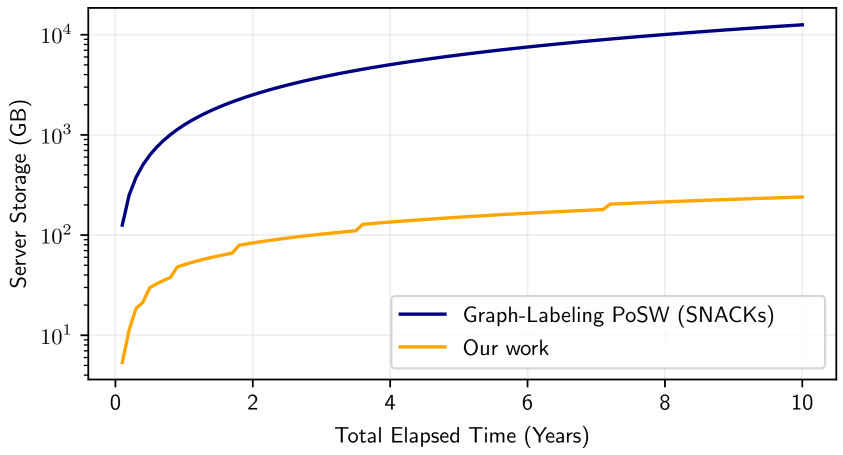


Figure 6: Prover Storage of SNACKs [7] and our InPoSW with Concrete Parameters. Our construction achieves >22× storage reduction assuming the PoSW instances have been running for 10 years.

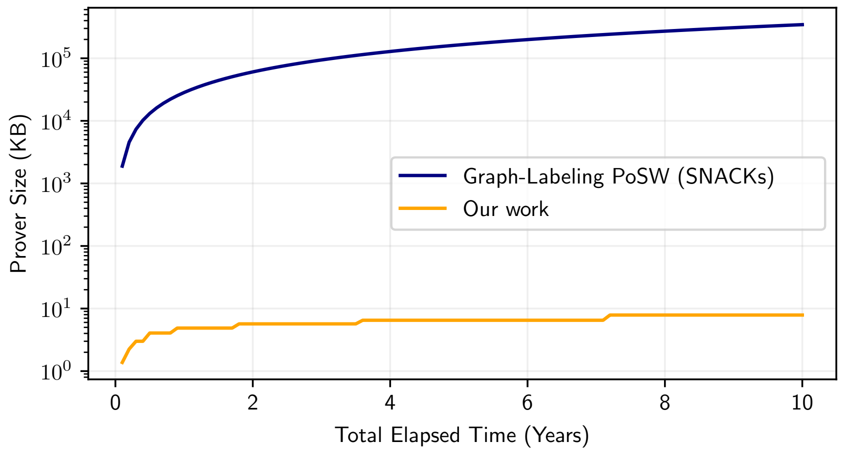


Figure 7: The Worst-Case Proof Size of SNACKs [7] and our InPoSW with Concrete Parameters. Our construction achieves >17900× proof size reduction assuming the PoSW instances have been running for 10 years.

computing $O(\lambda_H)$ exponentiation in a $O(\lambda_{\mathsf{VDF}})$-bit group during each verification. Although computational cost is dominated by $O(\log N \cdot \lambda_H)$ exponentiation, it is logarithmic to N and thus more efficient than SNACK verificationand is light-client friendly.

Concrete Parameters. Ethereum serves as our reference for estimation. For our scheme, we set $\Delta_T \approx 3.6$ minutes, which is the estimated time for the computation of 2^{33} squares[2] and allows timestamping of every new checkpoint upon its arrival. Ethereum emits finality checkpoints around every 6 minutes. Also, since the estimated time for $\geq 33\%$ validators to withdraw and become unstaked is at least 189 days [2, 1], an adversary cannot corrupt the finality of Ethereum within 189 days. While we can increase Δ_T to achieve a smaller proof size, we keep it to 3.6 minutes for generality purposeswith other PoS blockchains.We pick $\lambda_{\mathsf{VDF}} = 2048$ as suggested in [30], and we also set $\lambda_H = 256$ as in most blockchain systems. For SNACKs, we adopt RandomX [5] as the hash function for PoSW, which is ASIC-resistant and optimized for general-purpose CPUs. RandomX also has a much lower hash rate[3] than other commonly used hash functions, *e.g.*, SHA256. It is to SNACKs' advantage because it implies less prover storage and smaller proofs. We also set the soundness parameter as $\epsilon = 2^{-40}$ for SNACKs' verification.

Augmented Security. As explained above, a posterior long-range attack is only possible 189 days after a signed checkpoint is produced. With a VDF where an adversary can at best evaluate 5% faster[4], our protocol can extend the uncorrupted period to $189 \cdot 20$ days, equivalent to >10 years. For a VDF where an adversary might compute 1% faster, the extended uncorrupted period is >51 years. Therefore, even in the worst case, our protocol instantiated over the current state of Ethereum would provide protection for a sufficiently long period in the past.

Concrete Computation Cost. When our schemes have been running for 10 years, our construction needs ≈ 20 parallel VDF instances. Thus, the prover might need ≈ 20 processors for evaluation, which is well within the reach of GPUs or FPGA. For the verifier, the verification is mostly computing group exponentiation and multiplication over ≈ 20 sets of group elements, which can be efficiently done with a mobile device in a few seconds.

Concrete Prover Storage. Figure 6 plots the estimated prover storage of SNACKs and our InPoSW with [30] using the parameters above. When the system has been running for 10 years, SNACKs requires storing $\approx$ 12TB, while our scheme requires $\approx$ 546GB, achieving an order of magnitude ($> 22\times$)reduction.

Concrete Proof Size. Figure 7 shows the estimated worst-case proof size of SNACKs and our InPoSW with [30]. When the system has been running for 10 years, sending a SNACKs proof takes $\approx$ 337MB, while our scheme's proof takes only $\approx$ 20KB, which is $>17969\times$ reduction in communication cost.

7 Related Works

Non-Time-Dependent Protection. Winkle [10] requires changes to the consensus protocol, the delegate mechanism (to resolve slow converge issue) incurs a degree of centralization, and no external resource is involved to circumvent the impossibility result.

[2]This estimation bases on the best figures in VDF Alliance's VDF contest for optimizing hardware for repeated squaring.

[3]The best-known CPU chip can generate $\approx$ 1300 sequential hashes per second [4]

[4]VDF Alliance Competition shows that the performance of the winner's implementation and the first runner-up's differ by <4.5%.

Timestamps as Protection. Solana blockchain timestamps all its blocks and names it consensus protocol as proof-of-history. It produces a new block by hashing the previous block together with the new transactions, with the shortest possible interval between hashes like PoSW. However, it requires changes to the underlying PoS consensus protocol. Also, it is not light-client friendly because to verify a blockchain that has gone through T units of time, the client needs to verify an $O(\sqrt{T})$-size proof with T total computation. In contrast, our framework does not need to change the PoS consensus protocol, and the client only verifies $O(\text{polylog}\, T)$-size proof with $O(\text{polylog}\, T)$ total computation.

Both Babylon [28] and Pikachu [11] timestamp checkpoints by putting them on an external proof-of-work blockchain such as Bitcoin. However, its security inherently relies on the security of Bitcoin's PoW consensus. More importantly, this solution conflicts with the vision that most PoS blockchains will become the mainstream cryptocurrencies, eventually replacing the energy-inefficient Bitcoin. If such a vision becomes a reality, the benefits of compromising Bitcoin for long-range attacks may outweigh the cost of launching a 51% attack on Bitcoin's proof-of-work.

PoSAT [17] requires validators to evaluate VDFs and use the results to determine who will be the leader in producing new blocks. The VDF evaluations guarantee the elapsed time of the blockchain, making an adversary hard to fork another valid chain within substantially less time. However, this scheme requires changes to the consensus protocols.

Finally, Ethereum's finality gadgets [3], which require checkpoint signing from validators, is an imperfect solution, as they do not provide sufficient security against posterior corruption.

Multi-Resource Blockchain Protocol. Minotaur [20] integrates two distinct types of resources to construct a multi-resource blockchain. At first glance, the protocol appears resilient to long-range attacks as it harnesses both external (computational) and internal (stake) resources [9], with the computational aspect seemingly offering defense against such attacks. Nonetheless, a long-range attack remains feasible if an adversary can posteriorly corrupt a large portion of the stake and a small portion of the mining power.

PoSW in Blockchains. SNACK [7] leverages PoSW for succinct non-interactive proofs for PoW light clients. However, SNACK differs from our framework in many aspects. First, SNACK does not timestamp the blocks. Instead, it only proposes a generic framework to transform any graph-based PoSW sub-sampling technique to PoW light-client proof construction. Second, as SNACK does not timestamp the blockchain, it does not provide any security guarantee against long-range attacks. Third, our framework's PoSW is more efficient for aggregated timestamping.

Insertable PoSW and VDFs. Our performance estimation also shows that our scheme outperforms SNACKs by orders of magnitude. Other graph-labeling PoSW [16, 25, 18, 6] are subsumed by SNACKs in terms of succinctness of verification. Incremental PoSW [6, 18] reduces prover storage to polylogarithmic to the elapsed time but their proof size is very large as they require an excessive number of samples. Vanilla VDFs [12, 27, 27] do not support insertion and ever-going timestamps. Continuous VDF [19] supports ever-going timestamps but is not insertable. Also, it relies on specific hardness assumptions, while our construction uses VDFs as a black box regardless of their assumptions. Using dedicated VDFs for each timestamped data may achieve minimal proof size but will impose unbearable computational and storage costs on the server because these costs grow linearly with the number of inserted data times elapsed time.

Table 2 summarizes our comparisons with other long-range defense systems and related works.

8 Conclusion

To defend against long-range attacks in PoS blockchains, we propose Cumulus, which utilizes the cost-efficient timestamp service powered by our new primitive InPoSW. Our cost estimation shows that our InPoSW can reduce the prover's storage and the verification cost by orders of magnitude. More importantly, our defense system can be plugged into any PoS blockchain system for light-client without changing its underlying consensus mechanism.

Table 2: Comparison of our framework with Related Works

	Winkle	Solana	SNACK	Babylon/Pikachu	Minotaur	ETH's Finality Gadgets	PoSAT	**This work**
Timestamps as a Protection	✗	✓	✗	✓	✗	✗	✓	✓
No Hard Fork to Adopt	✗	✗	N/A	✓	✗	✗	✗	✓
No Centralized Server	✓	✗	✓	✓	✗	✗	✓	✓
Light-Client Friendly	✓	✗	✓	✓	✗	✓	N/A	✓
Energy-efficient	✓	✓	✓	✗	✓	✓	✓	✓

Disclaimers

References

[1] Ethereum validator queue, `https://www.validatorqueue.com/`

[2] Ethereum withdrawals: A comprehensive faq, `https://figment.io/insights/ethereum-withdrawals-a-comprehensive-faq/`

[3] Hierarchical finality gadget, `https://ethresear.ch/t/hierarchical-finality-gadget/6829`

[4] Randomx benchmark, `https://xmrig.com/benchmark/1`

[5] Randomx github repository, `https://github.com/tevador/RandomX`

[6] Abusalah, H., Cini, V.: An incremental posw for general weight distributions. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 282–311. Springer (2023)

[7] Abusalah, H., Fuchsbauer, G., Gaži, P., Klein, K.: SNACKs: Leveraging proofs of sequential work for blockchain light clients. Cryptology ePrint Archive, Report 2022/240 (2022), `https://eprint.iacr.org/2022/240`

[8] Abusalah, H., Kamath, C., Klein, K., Pietrzak, K., Walter, M.: Reversible proofs of sequential work. In: EUROCRYPT (2019)

[9] Azouvi, S., Cachin, C., Le, D.V., Vukolic, M., Zanolini, L.: Modeling resources in permissionless longest-chain total-order broadcast. In: OPODIS. LIPIcs, vol. 253, pp. 19:1–19:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)

[10] Azouvi, S., Danezis, G., Nikolaenko, V.: Winkle: Foiling long-range attacks in proof-of-stake systems. Cryptology ePrint Archive, Report 2019/1440 (2019)

[11] Azouvi, S., Vukolic, M.: Pikachu: Securing pos blockchains from long-range attacks by checkpointing into bitcoin pow using taproot. CoRR **abs/2208.05408** (2022)

[12] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 757–788. Springer, Heidelberg (Aug 2018)

[13] Bünz, B., Kiffer, L., Luu, L., Zamani, M.: FlyClient: Super-light clients for cryptocurrencies. In: 2020 IEEE Symposium on Security and Privacy. pp. 928–946. IEEE Computer Society Press (May 2020). https://doi.org/10.1109/SP40000.2020.00049

[14] Buterin, V., Griffith, V.: Casper the friendly finality gadget (2019)

[15] Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 451–467. Springer (2018)

[16] Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 451–467. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_15

[17] Deb, S., Kannan, S., Tse, D.: Posat: Proof-of-work availability and unpredictability, without the work. In: Borisov, N., Díaz, C. (eds.) FC (2021)

[18] Döttling, N., Lai, R.W.F., Malavolta, G.: Incremental proofs of sequential work. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 292–323. Springer, Heidelberg (May 2019)

[19] Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 125–154. Springer, Heidelberg (May 2020)

[20] Fitzi, M., Wang, X., Kannan, S., Kiayias, A., Leonardos, N., Viswanath, P., Wang, G.: Minotaur: Multi-resource blockchain consensus. Cryptology ePrint Archive, Report 2022/104 (2022), `https://eprint.iacr.org/2022/104`

[21] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 357–388. Springer, Heidelberg (Aug 2017)

[22] Lai, R.W.F., Malavolta, G.: Lattice-based timed cryptography. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO (2023)

[23] Lewis-Pye, A., Roughgarden, T.: Permissionless consensus. arXiv preprint arXiv:2304.14701 (2023)

[24] Mahmoody, M., Moran, T., Vadhan, S.P.: Publicly verifiable proofs of sequential work. In: Kleinberg, R.D. (ed.) Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013. pp. 373–388. ACM (2013)

[25] Mahmoody, M., Moran, T., Vadhan, S.P.: Publicly verifiable proofs of sequential work. In: Kleinberg, R.D. (ed.) ITCS 2013. pp. 373–388. ACM (Jan 2013)

[26] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), `http://bitcoin.org/bitcoin.pdf`

[27] Pietrzak, K.: Simple verifiable delay functions. In: Blum, A. (ed.) ITCS 2019. vol. 124, pp. 60:1–60:15. LIPIcs (Jan 2019)

[28] Tas, E.N., Tse, D., Yu, F., Kannan, S.: Babylon: Reusing bitcoin mining to enhance proof-of-stake security. Cryptology ePrint Archive, Report 2022/076 (2022)

[29] Tas, E.N., Zindros, D., Yang, L., Tse, D.: Light clients for lazy blockchains. arXiv preprint arXiv:2203.15968 (2022)

[30] Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part III. LNCS, vol. 11478, pp. 379–407. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17659-4_13

A Formal Proofs

In this section, we provide the formal proofs of our Theorems outlined in Sections 4 and 5.

Proof of Theorem 1. We provide the following Lemmas each proving completeness, soundness and succinctness respectively.

Lemma 1 *The honest verifier V outputs 0 (reject) only if a* VDF.Verify *or a* VecCom.Verify *outputs 0. It suffices to show that none of them will output 0 if the prover P is honest.*

- ***No*** VDF.Verify ***outputs*** 0. *In* InPoSW.Verify, *P sends the proof and output of $\mathcal{VDF}[i][j/2^i - 1]$ (for some i such that $t < 2^i \leq j$ and 2^i divides j) to V to verify. We can rewrite the index $j/2^i - 1$ to $(j - 2^i)/2^i$.*

  From Figure 4, we know that InPoSW.Eval *starts executing $\mathcal{VDF}[i][N'/2^i]$ at $N' \cdot \Delta_T$ (for some N') and this VDF should be finished after $2^i \cdot \Delta_T$, i.e., at $N' \cdot \Delta_T + 2^i \cdot \Delta_T = (N' + 2^i) \cdot \Delta_T$.*

  Plugging in $N' = j - 2^i$ and combining it with the fact that $j \leq N$, we know InPoSW.Eval *is finished with a proof by time $N \cdot \Delta_T$. Thus, P can provide V with a valid proof to pass* VDF.Verify.

- ***No*** VecCom.Verify ***outputs*** 0. *There are two locations where V invokes* VecCom.Verify. *We will inspect them separately.*

  For a VecCom.Verify *within the while loop, V checks if P can open the prior* vc' *(retrieved from $\mathcal{VC}[j]$), which is the input to a VDF starting time $j \cdot \Delta_T$, to y. Note that y is the output of the just verified VDF stored in $\mathcal{VDF}[i][j/2^i - 1]$, supposed to finish at $(j/2^i - 1) \cdot \Delta_T + 2^i \Delta_T = j \cdot \Delta_T$. According to Figure 4, P has committed y to $\mathcal{VC}[j]$. Thus, P can successfully open* vc' *to y.*

  For the final VecCom.Verify *in Figure 5 , V checks if P can open $\mathsf{vc}' = \mathcal{VC}[j]$ to $H(x)$. Because the while loop ends when $j = t$, and because x was inserted at $(N - k) \cdot \Delta_T = t \cdot \Delta_T$ as $\mathsf{aux} = H(x)$, P can successfully open it.* □

Lemma 2 *Here, we call the adversary by P because it takes the role of the prover, despite of acting maliciously, and interacts with V. At a very high level, we show that P must have executed VDFs with $(1-\epsilon)\cdot k\cdot \Delta_T$ sequential execution time and that x was inserted to the VDF at the beginning.*

First, P must have sequentially executed the VDFs checked in the while loop, and for any two adjacent VDFs verified in the loop, the VDF started later has to wait for the previously started VDF to finish. When P starts evaluating the later VDF, say, at time $j \cdot \Delta_T \cdot (1 - \epsilon)$, this VDF takes at input $\mathsf{vc} = \mathcal{VC}[j]$. As P has to open vc *to y at a fixed position i, P has to compute y by finishing the previous $\mathcal{VDF}[i][j/2^i - 1]$; otherwise, P either breaks the sequentiality of* VDF *to "predict" y or breaks the binding properties to open* vc *to y at a fixed position.*

Secondly, the total sequential execution of these VDFs (which are checked in the while loop) is $k\cdot\Delta_T\cdot(1-\epsilon)$. Every time j decreases by 2^i (for some $i \in \mathbb{N}$), V verifies if the checked VDF has sequential execution time

$2^i \cdot \Delta_T \cdot (1-\epsilon)$; otherwise, it breaks the sequentiality of the VDF set up with $\mathsf{VDF.Setup}(\lambda_{\mathsf{VDF}}, \Delta_T \cdot 2^i)$. *As j decreases from N to $t = N-k$, the total sequential execution time is $(N-(N-k)) \cdot \Delta_T = k \cdot \Delta_T \cdot (1-\epsilon)$.*

Finally, P must have inserted x into the input commitment of the first VDF in the above VDF sequence, i.e., $\mathcal{VC}[t]$, which started at $k \cdot \Delta_T$ ago. If x was not inserted but P can open the inputted vector commitment to $H(x)$, P breaks the binding property of VecCom *or the collision resistance of the hash function.*

As a PPT adversary can only break the sequentiality of VDF, *the binding properties of* VecCom, *and the collision resistance of the hash function with negligible probability, P can only break the soundness with negligible probability.* □

Lemma 3 *The communication and computation cost counted in Figure 5 shows* InPoSW.Verify *imposes only $O(\mathsf{poly}(\lambda, \log T, \log N\Delta_T))$ cost to the verifier.* □

Proof of Theorem 2. We prove it by contradiction. Assume an adversary $\mathcal{A}$ who is under the security assumption Π and can break the $T_\Pi \cdot (1/\epsilon)$-extended-finality security even if Π is coupled with BootBiPoSW. Then, $\mathcal{A}$ will can also break T-uncorrupted-finality of Π: As $\mathcal{A}$ can break the $T_\Pi \cdot (1/\epsilon)$-extended-finality security, then for a verifier V and an honest prover P with a blockchain BC_{honest} and prepared bootstrap state st_{honest}, $\mathcal{A}$ can produce some forks $\{BC_i\}_i$ and bootstrap state $\{st_i\}_i$ such that after running BootBiPoSW.Verify, V obtains a commitment vc that is not committed to BC_{honest}. There are only two cases where BootBiPoSW.Verify outputs a wrong commitment.

- **Case 1.** BiGame outputs a commitment set of the blockchain but none of the commitment inside is committed to BC_{honest}. By the completeness of BiGame, it only happens with negligible probability.
- **Case 2.** A commitment of BC_{honest} is in BiGame's commitment set but not in the final commitment set of BootBiPoSW. It happens only if $\mathcal{A}$ provides a $BC_{\mathcal{A}}$ such that $BC_{\mathcal{A}}[\mathsf{prefix}+1]$ should be chosen over $BC_{\mathsf{honest}}[\mathsf{prefix}+1]$ and also provided a InPoSW timestamp proof for $BC_{\mathcal{A}}[\mathsf{prefix}+1]$ showing that $T_{\mathcal{A}} \cdot \Delta_T \geq T_{\mathsf{honest}} \cdot \Delta_T = T_{\mathsf{real}}$, where T_{real} is the elapsed time from the checkpoint is produced until the verification. However, due to T_Π-uncorrupted-finality of Π, $\mathcal{A}$ has to start producing the VDF output and proof T_Π later than the honest server. Due to the $(1-\epsilon) \cdot t$-sequentiality of the VDF, $\mathcal{A}$ cannot acquire the output and proof in time if $T_\Pi + (1-\epsilon) \cdot T_{\mathcal{A}} \cdot \Delta_T \leq T_{\mathsf{honest}} \cdot \Delta_T$ except for negligible probability. It implies $T_\Pi + (1-\epsilon) \cdot T_{\mathsf{real}} \leq T_{\mathsf{real}}$ (by replacing $T_{\mathcal{A}} \cdot \Delta_T$ and $T_{\mathsf{honest}} \cdot \Delta_T$ by T_{real}), and thus $T_\Pi \cdot (1/\epsilon) \leq T_{\mathsf{real}}$. Hence, after the checkpoint being produced, $\mathcal{A}$ has to wait until $T_\Pi \cdot (1/\epsilon)$ of time passes, which is our extended uncorrupted finality. Or, $\mathcal{A}$ breaks the soundness of InPoSW, which happens with only negligible probability.

B Formal Definition

Definition 5 (Bootstrap Protocol) *A bootstrap protocol for some blockchain scheme Π is a protocol* Bootstrap *where n provers $P_1, \ldots, P_n$ interact with a verifier V. Each prover P_i has knowledge of a chain BC_i and a commitment vc_i of BC_i. After the interaction, the verifier outputs a index of the prover i and accepts this prover's commitment vc_i. Assuming at least one of the provers is honest,*

- ***Completeness**: With overwhelming probability, if a prover P_i is honest, and the protocol execution outputs a commitment vc_k, then vc_i and vc_k should open to the same blockchain BC.*
- ***Soundness**: With overwhelming probability, if there is a chain BC_j chosen over BC_i w.r.t.* Π, *then i should not be chosen as the output index.*

Definition 6 (Verifiable Delay Function) *A* $\mathsf{VDF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ *is a triple of algorithms as follows:*

- $\mathsf{Setup}(\lambda, t) \to \mathsf{pp} = (\mathsf{ek}, \mathsf{vk})$ *where λ is the security parameter, t is the desired puzzle difficulty that specifies the elapsed time intended to prove, and* ek *and* vk *are the evaluation key and verification key, respectively. (The rest of the algorithms implicitly take* pp *as input.)*

- Setup:
  - Input: P inputs a security parameter λ.
  - Output: V receives a state st_1
  - Procedure:
    1. P stores λ and implicitly uses it in other procedures.
    2. P uses λ and Π (the blockchain scheme) to decides Δ_T
    3. P invokes $\mathsf{st}_{\mathsf{InPoSW}} = \mathsf{InPoSW.Gen}(\lambda, \Delta_T)$
    4. P fetches the latest blockchain BC and compute $\mathsf{Bs} = (B_i)_i$
    5. P computes $\mathsf{vc} = \mathsf{VecCom.Com}(\mathsf{Bs})$ and $\mathsf{st}'_{\mathsf{InPoSW}} = \mathsf{InPoSW.Eval}(\mathsf{st}'_{\mathsf{InPoSW}}, \mathsf{vc})$
    6. P outputs $\mathsf{st} = (BC, \mathcal{VC} = (\mathsf{vc}), \mathsf{st}'_{\mathsf{InPoSW}})$
- Prepare:
  - Parameters: P has λ and Δ_T.
  - Input: P inputs its previous state st_N
  - Output: V receives the updated state st_{N+1}
  - Procedure:
    1. P parses $(BC, \mathcal{VC}, \mathsf{st}'_{\mathsf{InPoSW}}) = \mathsf{st}_N$ and fetches the latest blockchain BC'
    2. If $BC' \neq BC$, P compares them, retrieves the new blocks $\mathsf{Bs} = (B_i)_i$ and computes $\mathsf{vc} = \mathsf{VecCom.Com}(\mathsf{Bs})$; otherwise, $x = \bot$
    3. P computes $\mathsf{st}'_{\mathsf{InPoSW}} = \mathsf{InPoSW.Eval}(\mathsf{st}'_{\mathsf{InPoSW}}, \mathsf{vc})$
    4. P stores $\mathsf{st}_{N+1} = (BC', \mathcal{VC}||\mathsf{vc}, st'_{PoSW})$
- Verify:
  - Parameters: All prover $\{P_i\}$ and the verifier V agree on λ, Δ_T, and also T_Π, the uncorrupted finality of Π.
  - Input:
    - Each P_i inputs its state st_i
    - V inputs nothing
  - Output: V outputs i and vc, which are the index of the prover and the vector commitment of a blockchain that V accepts, respectively.
  - Procedure:
    1. Each P_i parses its st_i to get BC_i and sends to V its $\mathsf{vc}_i = \mathsf{VecCom}(\mathsf{Bs}_i)$, where $\mathsf{Bs} = (B_j)_j$ in BC_i.
    2. V checks if all $\{\mathsf{vc}_i\}_i$ are the same. If so, outputs any i and vc_i and halt.
    3. V gets $(I_{\mathsf{BiGame}}, \mathsf{prefix}) = \mathsf{BiGame}(BC_1, \ldots, BC_m)$ by interacting with all $\{P_i\}_i$
    4. If $\mathsf{vc}_i = \mathsf{vc}_j$ for all $i, j \in I_{\mathsf{BiGame}}$, V outputs (i, vc_i) by randomly choosing a $i \in I_{\mathsf{BiGame}}$ otherwise, proceeds to the next step.
    5. For each $i \in I_{\mathsf{BiGame}}$, V asks P_i for the finality proof $BC_i[\mathsf{prefix}+1].\mathsf{FinalPf}$ at the first differing block.
    6. Iterating through each $i \in I_{\mathsf{BiGame}}$, if V finds that for the current i:
       - $\mathsf{InPoSW.Verify}(\mathsf{st}_i.\mathsf{st}_{\mathsf{InPoSW}}, t_i, \mathsf{vc}_i) \overset{?}{=} 1$ and vc_i can be opened to $B_i[\mathsf{prefix}+1].B$
       - no other all other $j \in I_{\mathsf{BiGame}}$ such that $BC_j[\mathsf{prefix}+1]$ should be chosen over $BC_i[\mathsf{prefix}+1]$ and $\mathsf{InPoSW.Verify}(\mathsf{st}_j.\mathsf{st}_{\mathsf{InPoSW}}, t_j, \mathsf{vc}_j) \overset{?}{=} 1$ and vc_j can be opened to $B_j[\mathsf{prefix}+1].B$ and $t_j\Delta_T \leq (t_i+1)\cdot\Delta_T + T_\Pi$.

       then V outputs i and vc_i and halts.

Figure 8: Construction of our Bootstrap Protocol BootBiPoSW for augmenting a PoS blockchain scheme Π

- $\mathsf{Eval}(x) \rightarrow (y, \pi)$ *takes an input x and produces an output y and a proof π.*
- $\mathsf{Verify}(x, y, \pi) \rightarrow 0/1$ *is a deterministic algorithm which takes an input x, output y and proof π and outputs 1 or 0.*

VDF *should satisfy the following properties:*

- ***Correctness.*** *A VDF V is correct if for all λ, t parameters $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{Setup}(\lambda, t)$, and all $x \in X$, if $(y, \pi) \leftarrow \mathsf{Eval}(ek, x)$ then $\mathsf{Verify}(x, y, \pi) = 1$.*
- ***Soundness.*** *A VDF is sound if for all algorithms $\mathcal{A}$ that run in time $O(poly(t, \lambda))$,*

$$\Pr\left[\mathsf{Verify}(x, y, \pi) = 1 \;\middle|\; \mathsf{pp} = (ek, vk) \leftarrow_R \mathsf{Setup}(\lambda, t)\right] \leq negl(\lambda)$$

- $\sigma(t)$***-Sequentiality.*** *For a function $\sigma(t)$ and given a randomly chosen x and $pp \leftarrow_R \mathsf{Setup}(\lambda, t)$, no adversary with $\mathsf{poly}(\lambda)$ processors can produce, in parallel time $\sigma(t)$, an output y_A such that $y_A = y$ where $(y, \pi) := \mathsf{Eval}(pp, x)$ with non-negligible probability.*
- ***Succinctness.*** *Algorithm* Verify *must run in total time polynomial in $\log t$ and $O(\lambda)$. Notice that* Verify *is much faster than* Eval.

C More Definitions

To be self-contained, we provides the definitions of vector commitments and bisection games. Yet, our protocol only uses a subset of the functionalities of vector commitments, and Merkle Tree suffices for our uses.

Definition 7 (Vector Commitment) VecCom *consists of the following functionality.*

- $\mathsf{VecCom.Setup}(1^\lambda, q)$*: Given the security parameter k and the size q of the committed vector (with $q = poly(k)$), the key generation outputs some public parameters pp (which implicitly define the message space M).*
- $\mathsf{VecCom.Com}(m_1, \ldots, m_q)$ *: On input a sequence of q messages $m_1, \ldots, m_q \in M$ and the public parameters pp, the committing algorithm outputs a commitment string C and an auxiliary information aux.*
- $\mathsf{VecCom.Open}(m, i)$ *: This algorithm is run by the committer to produce a proof π_i that m is the i-th committed message. We also implicitly let the prover to read its auxiliary data.*
- $\mathsf{VecCom.Verify}(\mathsf{vc}, m, i, \pi_i)$*: The verification algorithm accepts (i.e., it outputs 1) only if π is a valid proof that C was created with a sequence $m_1, \ldots, m_q$ such that $m = m_i$.*

They satisfies the following properties:

- ***Position Binding.*** *A vector commitment satisfies position binding if for all $i = 1, \ldots, q$ and for every PPT adversary $\mathcal{A}$, the following probability (which is taken over all honestly generated parameters) is at most negligible in λ:*

$$\Pr\left[\begin{matrix}\mathsf{VecCom.Verify}(\mathsf{vc}, m, i, \pi) = 1 \wedge \\ \mathsf{VecCom.Verify}(\mathsf{vc}, m', i, \pi') = 1 \wedge m \neq m'\end{matrix} \;\middle|\; (C, m, m', i, \pi, \pi') \leftarrow \mathcal{A}(pp)\right]$$

- ***Succinctness:*** *The computation cost of* VecCom.Verify *should be $O(\mathsf{poly}(\log m, \lambda))$. The size of a vector commitment* vc *and the proof π for openings should be of $O(\mathsf{poly}(\log m, \lambda))$ size.*

Definition 8 *A bisection game is a protocol* BiGame *that includes a number of provers* $P_1, \ldots, P_n$ *providing its chain* $BC_1, \ldots, BC_m$*, respectively, and a verifier* V*. Running* $\mathsf{BiGame}(BC_1, \ldots, BC_m)$ *with the provers, the verifier receives an index set* I *and an block index* prefix*. Assuming at least one of the provers is honest, and at least one of* BC *is a valid blockchain,* BiGame *satisfies the following properties:*

- ***Completeness****: With overwhelming probability, if all blocks and states in* BC_i *are valid, then* $i \in I$*. In addition,* prefix *is the maximum* j *such that a block* B_j *and state* st_j *exists in all* BC_i *for* $i \in I$*.*
- ***Soundness****: If* BC_i *is not a valid chain, then* $i \notin I$*.*