stoRNA: Stateless Transparent Proofs of Storage-time

Reyhaneh Rabaninejad[1], Behzad Abdolmaleki[2], Giulio Malavolta[3], Antonis Michalas[1,4], and Amir Nabizadeh

[1] Tampere University, Finland
{reyhaneh.rabbaninejad, antonios.michalas}@tuni.fi
[2] University of Sheffield, United Kingdom
behzad.abdolmaleki@sheffield.ac.uk
[3] Max Planck Institute for Security and Privacy, Germany
giulio.malavolta@mpi-sp.org
[4] RISE Research Institutes of Sweden

Abstract. Proof of Storage-time (PoSt) is a cryptographic primitive that enables a server to demonstrate non-interactive continuous availability of outsourced data in a publicly verifiable way. This notion was first introduced by Filecoin to secure their Blockchain-based decentralized storage marketplace, using expensive SNARKs to compact proofs. Recent work [2] employs the notion of trapdoor delay function to address the problem of compact PoSt without SNARKs. This approach however entails statefulness and non-transparency, while it requires an expensive pre-processing phase by the client. All of the above renders their solution impractical for decentralized storage marketplaces, leaving the stateless trapdoor-free PoSt with reduced setup costs as an open problem. In this work, we present *stateless* and *transparent* PoSt constructions using probabilistic sampling and a new Merkle variant commitment. In the process of enabling adjustable prover difficulty, we then propose a multi-prover construction to diminish the CPU work each prover is required to do. Both schemes feature a fast setup phase and logarithmic verification time and bandwidth with the end-to-end setup, prove, and verification costs lower than the existing solutions.

1 Introduction

Storage-as-a-Service, including cloud storage services and, more recently, Decentralized Storage Networks (DSNs) [22, 16], has attracted extensive interest and caused big data migration from local storage systems to storage servers, as it offers efficient and scalable services at a lower cost. However, after outsourcing, the data owner has no physical control over the data. Hence, continuous data availability is an important trait that highly-reliable service providers [5] should guarantee to protect users against downtime, whatever its cause, and ensure that data owners can retrieve their data files at any time. Continuous data availability is becoming increasingly critical as it provides global ceaseless access to online

business data and business-to-business applications. The existing notion of Proof of Storage (PoS) [1, 11] ensures data integrity and availability at a specific time point (i.e., the time the challenge is issued). A naive approach to certify continuous data availability consists of using PoS and performing frequent checks over time. However, this requires that clients be online when sending sequential challenges to the storage server. Moreover, in DSNs such as Filecoin [16], where proofs are verified by the blockchain network, this method causes communication complexities and, eventually, leads to network bottlenecks.

1.1 Proof of Storage-time

Ateniese *et al.* [2] formalized the notion of Proof of Storage-time (PoSt) to address the issue of continuous availability guarantees for outsourced data, and proposed two constructions in the random oracle model. Informally, a PoSt protocol enables storage servers to efficiently convince a verifier that data is continuously available and retrievable via generating chained sequential challenge-responses over a specified time interval. Consider D as the time period during which a specific data file is deposited in the server. D is divided into time slots of length T, where T is the audit frequency parameter – the prover is challenged once in every time slot T, while the verifier is not required to remain online. This helps approximating continuous data availability throughout a D time range with discretized frequent auditing, where a smaller T provides a superior availability guarantee. The measure of time here is the number of unit steps of the Turing machine. Let *timer* be a global (verification) timer initiated by the data owner, but public (with the timer the verification algorithm can check whether the final proof is received on time). A PoSt consists of a tuple of four algorithms $\mathsf{PoSt} = (\mathsf{Setup}, \mathsf{Store}, \mathsf{Prove}, \mathsf{Verify})$, as defined below.

- $\mathsf{Setup}(1^\lambda, T, D) \rightarrow (\mathsf{par}, \mathsf{sk})$: Inputs security parameter λ, audit frequency parameter T, D and outputs the public parameters par and secret key sk.
- $\mathsf{Store}(F^*, \mathsf{sk}, \mathsf{par}, T, D) \rightarrow (F, \mathsf{tg})$: Takes as input an original data file F^*, a secret key sk, an audit frequency parameter T, and a deposit time D and generates an encoded file F. It also outputs tag tg as necessary information to run $\mathsf{PoSt.Prove}$ and $\mathsf{PoSt.Verify}$ algorithms.
- $\mathsf{Prove}(\mathsf{par}, chal, \mathsf{tg}, F) \rightarrow \pi$: Inputs encoded file F, tag tg, public parameters par, and challenge seed *chal* issued by a verifier at the outset of the deposit period, and outputs proof π promptly after the deposit period ends.
- $\mathsf{Verify}(\mathsf{par}, \mathsf{sk}, \mathsf{tg}, chal, \pi, timer) \rightarrow \{accept, reject\}$: Inputs par, secret key sk, tag tg, challenge *chal*, proof π, and *timer* to check timely reception of the final proof. It outputs a bit b to designate *accept* or *reject*.

PoSt schemes may present the following core features:

Public Verifiability. A smart contract or any third party (not just the clients) are able to audit continuous data availability by verifying the output from $\mathsf{PoSt.Prove}$ algorithm. To this end, the verification algorithm $\mathsf{PoSt.Verify}$ should not take any secret sk as input.

Statelessness. An unbounded (polynomial) number of verifications are supported without requiring the verifier to maintain the protocol state. If a PoSt protocol is stateful, when the number of verifications reaches a pre-determined fixed bound, the protocol stops and no further audits are possible, unless the data owner retrieves outsourced files and relaunches the $\mathsf{PoSt.Store}$ algorithm.

Dynamic. Efficient updates on outsourced data are enabled at any time without the need for an expensive setup.

Transparency. A PoSt scheme may import a one-time trusted setup run by an honest client with a publicly published setup output to all entities. However, a PoSt scheme is transparent if its setup does not involve any secret sk. This property is necessary in DSNs where provers may also be clients and prevents *generation attack*– that is, a malicious client-prover output a valid proof at the time a challenge is issued by generating data *on-the-fly* to collect network rewards, without really reserving storage.

Compactness. Low verification cost is enabled independent of the file size and deposit length.

Additionally, a PoSt scheme must present the following security properties:

Completeness. For all files $F^* \in \{0,1\}^*$, all $(\mathsf{par}, \mathsf{sk})$ values output by $\mathsf{Setup}(1^\lambda, T, D)$, and all (F, tg) output by $\mathsf{Store}(F^*, \mathsf{sk}, T, D)$, a proof π generated by honest prover in $\mathsf{Prove}(\mathsf{par}, chal, \mathsf{tg}, F)$ on the challenge $chal$ will cause $\mathsf{Verify}(\mathsf{par}, \mathsf{sk}, \mathsf{tg}, chal, \pi, timer)$ to always output *accept*.

Soundness. This property guarantees that if a prover is able to convince an honest verifier that it has stored a file throughout the specified deposit time then there is an extractor Ext, that given a subset of prover configurations and the code of the transition function (i.e the random-coin r of the prover) can extract data via interacting with the prover[5]. Formally, a PoSt scheme is sound, if for any PPT adversary $\mathcal{A}$ for a file F, there is an extractor $\mathsf{Ext}_{\mathcal{A}}$ s.t for all λ and all files $F^* \in \{0,1\}^*$,

$$\Pr\left[\begin{array}{l}(\mathsf{par}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda, T, D); (F, \mathsf{tg}) \leftarrow \mathsf{Store}(F^*, \mathsf{sk}, T, D);\\ (\pi || \hat{F}) \leftarrow (\mathcal{A} || \mathsf{Ext}_{\mathcal{A}})(\mathsf{par}, chal, \mathsf{tg}, F; r) :\\ \mathsf{Verify}(\mathsf{par}, \mathsf{sk}, \mathsf{tg}, chal, \pi, timer) \wedge \hat{F} \neq F\end{array}\right] \approx_\lambda 0$$

Here, $chal$ is the verifier challenge and tg is a tag corresponding to file F.

Ateniese's et al. Construction in a Nutshell: The work in [2] presents two different constructions of PoSt. The first warm-up protocol is based on the intuition proposed in the Filecoin whitepaper [16]: the prover generates sequential Proofs of Retrievability (PoRs), where each PoR proof is computed based on a challenge derived from the PoR proof in a previous iteration. As a result, the verifier merely provides the first challenge and can then go offline. The scheme

[5] The data should be extracted from the configuration corresponding to any specific time and the transition function.

leverages the notion of Verifiable Delay Function (VDF) [6] to guarantee a specific amount of delay between two successive PoR proofs, even if the prover uses parallel processors. In every time slice, the prover evaluates VDF by feeding a priori PoR proof as its input and generates a challenge by hashing the VDF output. The prover returns all sequential challenge-proof pairs along with the respective VDF proofs, to be inspected all at once by the verifier. The verification procedure of this warm-up construction however is very expensive since the verifier must audit all proofs one-by-one, and the communication cost is high.

The second protocol follows a different approach based on the Trapdoor Delay Function (TDF). The client executes a pre-processing phase to generate a tag, producing the same sequence of challenge-proof pairs as the prover, but with faster TDF evaluations due to the trapdoor. Nevertheless, in this compact scheme the client relies on a trapdoor to run the setup phase and generate challenges, thus it does not provide public verifiability. Moreover, this protocol is stateful and static. Besides, the soundness of the compact scheme assumes the holder of the trapdoor is honest. This signifies that contrary to what is stated by the authors, the construction cannot be directly used in the DSNs as is the case with Filecoin. Authors have pointed several aspects that remain unresolved such as: (i) support for dynamic data updates, (ii) stateless and transparent PoSt constructions (without trapdoors), and (iii) setup cost reduction.

In light of these issues we ask the following question: *Can we have a Proof of Storage-time for continuous availability monitoring of dynamic data at storage providers in a transparent, stateless, yet efficient manner?*

1.2 Our Contributions

This work makes significant progress in answering the above question. We propose storNA, a new stateless PoSt protocol with a fast setup for light clients, aiming to outsource their data to a storage network for a deposit period. A public verifier can verify continuous data availability with computation and communication overheads logarithmic in the length of the deposit period. The construction can be instantiated from any stateless publicly verifiable PoR and invokes Proof of Elapsed-time (PoEt) proposed in [8] as a trust-less proxy for time.

Commitment and Random Sampling. We present a new commitment graph inspired by the Directed Acyclic Graph (DAG) introduced in [8], where every single graph node is efficiently updated in sequential time slots, based on the notion of the Merkle Mountain Range, and takes external inputs from the proofs generated at said time slot. The constant-size root of this graph plays as a commitment over the whole PoSt sequence generated by the prover. This commitment mechanism enables the verifier to randomly sample and verify only a logarithmic number of proofs from the PoSt sequence. Inclusion proofs of the commitment graph aid the verifier to check if the proofs returned by the prover are bound to challenged positions of the PoSt sequence.

Stateless-Transparent-Dynamic Construction. Since the client does not rely on any trapdoor, storNA is *transparent* and copes with malicious clients.

Moreover, it provides unbounded use: when number of verifications reaches an a priori bound (deposit period ends), the client can extend it with no computation (*deposit-extendability*). This is possible due to the incremental nature of the proof chain: the prover can keep up the chain from the last state to append further PoRs at agreed frequency. stoRNA also enables dynamic updates on outsourced files at marginal costs (*file-extendability*).

Multi-Prover Setting. stoRNA is in single-prover setting: the prover hosting the data and providing storage proofs also proves the passage of time between successive storage proofs. We next extend stoRNA to a multi-prover PoSt construction, mstoRNA, which differs as regards prover resources. Any arbitrary number of provers can join the decentralized market by providing their preferred resources: (i) *Time Nodes*, who mainly spend CPU work by continuously running PoEt and periodically publishing the publicly verifiable state, and (ii) *Storage Nodes* who provide storage-time by renting out disk-space over time to the clients. A PoSt sequence generated by a prover in this construction is like a public storage-ledger that any one in public can verify, while it can migrate to any other prover, who may continue the ledger where previous prover left off. This aspect is particularly important when considering the rapid-changing distributed nature of DSNs with real nodes susceptible to failure.

1.3 Technical Overview

Consider a data owner wishing to outsource its data to the storage provider(s) and verify continuous data availability without remaining online. Additionally, at a later time, the data owner may extend the deposit time or update its outsourced files without relaunching the entire setup or adding much cost to the verification algorithm. stoRNA enables any light client to do so: the client only requires to perform an efficient Store algorithm to generate necessary information for the prover and public verifier. Each storage provider, participating in the stoRNA protocol, stores the data file for a specific deposit period. To prove "storage-time" i.e., continuous availability of the specified storage over the specified deposit time, the storage provider sequentially generates PoRs during the entire storage period. To compel a specific amount of delay between successive PoRs generated by the prover, the protocol leverages the concept of publicly verifiable PoEt.

In order to enable efficient verification procedure with low communication, the verifier randomly samples and verifies a logarithmic number of proofs from the chain. However, with this probabilistic sampling approach, a dishonest prover can fool the verifier by sending correctly-generated proofs from *arbitrary* time slots in response to the verifier's challenge. Hence, sampling fails to catch continuous data availability with high probability.

One way to enforce the prover to send storage proofs at the *precise* challenged time slots on the chain, is to have him *commit* to the entire chain before random slots are sampled. As a result, the verifier can use the commitment to check whether the returned responses belong to the challenged slots. To commit to the whole chain of sequential proofs, the prover updates a graph G_n^{Com} based

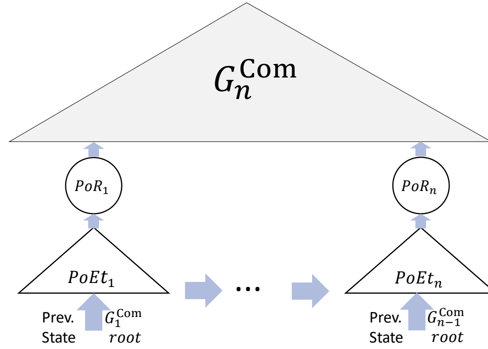


Fig. 1: Structure of storNA.

on a variation of the Merkle commitment with some extra edges as illustrated in Figure 2, across all PoRs generated up to the current time. This commitment graph G_n^{Com} is inspired by the elegant DAG proposed in [8], but with subtle modifications to be discussed in section 4. At each time slot i, the prover efficiently updates the G_n^{Com} by appending the hash of the most recent PoR and PoEt proofs to the labels of the parents of node i and uses the new tree root as a statement to run the next PoEt (Figure 1). Consequently, the tree root at each time slot i, plays as a commitment over the whole chain up to that slot.

At the end of deposit period, the prover returns the root label of latest G_n^{Com} as a commitment to the entire series of proofs generated within the whole period. Upon receiving the commitment, the verifier challenges a randomly sampled subset of time slots (this can be made non-interactive using the Fiat-Shamir heuristic [10] – i.e., the prover can generate challenge slots himself by hashing the commitment). For every sampled time slot, the prover provides the corresponding PoR, PoEt proofs together with the logarithmic size Merkle opening. The verifier, first checks whether returned proofs are located at the challenged positions of the chain previously committed by, which is made possible via the position-binding guarantee in the Merkle proofs. Next, it checks the correctness of the PoR, PoEt proofs. If the PoR, PoEt proofs or the Merkle opening of any sampled slot is invalid, the verifier will reject. Else, the verifier is conviced that the commitment is computed *mostly* correct. Figure 1 depicts a schematic overview of our construction, named storNA since the single strand built by the prover can be viewed as a **sto**rage **RNA**[6] that carries information about client data: the extractor algorithm of the underlying PoR scheme can use PoRs appended over time to the chain, to extract the client file with high probability.

Multi-Prover PoSt construction. The above construction, seems to provide all desirable features at once:

[6] RNA is a single strand biological molecule essential in coding, decoding, and expression of genes.

- *Fast setup and transparency.* The client only requires to perform the Store algorithm of the underlying PoR scheme on the data files before outsourcing without relying on any trapdoors. Hence, the scheme is transparent.
- *Logarithmic verification time and bandwidth.* The verifier algorithm can audit continuous data availability with high probability in time and communication logarithmic in the length of the deposit period.
- *Statelessness and unbounded use.* When the number of verifications reaches the a priori bound (deposit period ends), the client can easily extend the deposit period without relaunching the Store algorithm. The prover can keep up the PoSt sequence from the last state to append further PoRs at the agreed frequency.
- *Dynamic.* Assuming the underlying PoR scheme supports dynamic databases, the client can update its outsourced files at any time without a re-computation of the entire initialization algorithm. The client only requires to perform fast setup *on the modified data blocks* and outsources them to be updated at the storage server. The verifier needs to use the new PoR tags for auditing the chain from the point update takes place.

However, there is still a challenge to be addressed: Even the honest prover algorithm requires heavy inherently sequential CPU computations. More precisely, the prover participating in the network, needs to spend two distinct resources: (i) storage-time (storage resources over time) and (ii) CPU work (CPU power over time). The first one is natural in PoSt mechanisms as the prover has to dedicate a specified amount of disk-space over time. But the CPU work is due to the use of PoEt in the protocol to guarantee a delay between storage proofs and prevent the prover from generating all required proofs at once and discarding the data. This CPU work is a major deterrent to renting out storage by storage providers or leads to increased storage fees in decentralized storage markets.

Our second construction, mstoRNA (shown in appendix B for space constraint), is based on division of CPU work and storage-time resources between "Time Nodes"and "Storage Nodes". Time Nodes participate in the decentralized market by continuously running PoEt algorithm. At each time slot, the Time Node advertises the PoEt state and waits for Storage Nodes to submit PoR proofs generated based on the challenge derived from the freshly advertised PoEt state and the signature of each individual Storage Node. The wait time is specified based on the network roundtrip time (RTT). Next, the Time Node (i) creates a Merkle tree with the PoRs collected from Storage Nodes, (ii) inputs the Merkle root together with PoEt proof to update G_n^{Com}, and (iii) timestamps the updated commitment into the PoEt sequence by appending the most recent G_n^{Com} root into the shared PoEt state. At the end of the deposit period, the Time Node, acting as the main prover interacting with the verifier in this network, returns the root-label of the latest commitment graph as a commitment to all proofs from all Storage Nodes sequentially generated during the deposit period. Upon receiving the commitment, the verifier simply opens some of the committed labels to verify both storage and time proofs included in those labels. In Table 1, we give a high-level comparison of our constructions over compact PoSt [2].

Table 1: Comparison of our constructions over compact PoSt [2]. $N = \frac{D}{T}$ denotes number of iterations during the deposit period D, $t = \log \frac{T}{2}$, $n = \log \frac{N}{2}$, and m denotes the number of Storage Nodes connected to a Time Node in mstoRNA construction.

	Features			Overhead		
	stateless	transparent	dynamic	setup	verification	proof size
cPoSt [2]	✗	✗	✗	$O(N)$	$O(1)$	$O(1)$
stoRNA	✓	✓	✓	$O(1)$	$O(tn)$	$O(tn)$
mstoRNA	✓	✓	✓	$O(1)$	$\frac{O(tn)}{m}$ ‡	$\frac{O(tn)}{m}$ ‡

‡ These are with respect to a single Storage Node.

1.4 Application Domain

Here, we exemplify applications our stoRNA construction could be beneficial to.

Blockchain History Expiry. Hard disk storage is one of the biggest bottlenecks in L1 blockchain scalability. For example, the Ethereum chain will become gigantic in the coming years, making storage infeasible for individuals. The idea of History expiry is to obviate the need for all, full nodes to download the entire chain from genesis. Instead, only the most recent historical blocks would be held and served by the core blockchain protocol. Older blocks would be stored by external storage providers, which can minimize requirements for node hard drive space, paving the way for further decentralization. Many Decentralized Applications (DApps) are already removing data from blockchains for efficiency. However, since being an immutable trustless record is one of the principal features of blockchain, long term availability of older blocks should be guaranteed. Using our method one can publicly verify long term continuous availability of large historical blocks.

Decentralized Storage Networks. DSN is a decentralized algorithmic market based on blockchain made up of various nodes rewarded for storing and maintaining data availability. The network controls the accessible disk space, disperses client data across nodes, audits the integrity and retrievability of data, restores possible failures and rewards honest nodes. The stateless and transparent nature of stoRNA makes it suitable for audit purposes in DSNs.

2 Related Work

Proofs of Storage (PoS) schemes enable clients to outsource files to a server, and later in an interactive audit phase, verify the integrity of the stored data. A verifier, repeatedly challenges the server and checks the returned proof that the server is still storing the client's file intact. The term verifier refers to the client, who originally outsourced the file (privately verifiable PoS), or any third party (publicly verifiable PoS).These protocols are also known as Provable Data Possession (PDP) [3]. Proofs of Retrievability (PoR) schemes [11] are similar to PDP, but they additionally guarantee data retrievability, achieved by an extractor that reconstructs the client file from the proofs returned by the prover. The

extensive research on PDP/PoR schemes covers various advanced features including dynamic data updates [7], shared data files [17], and proof of replicated storage [18].

Proofs of Space (PoSpace) schemes enable a prover to convince a verifier certain disk space is dedicated. PoSpace schemes can be used as an alternative to the blockchain Proof of Work (PoW) consensus mechanism, where instead of the CPU computation, disk-space is expended [9]. PoSpace can also be viewed as a PoS scheme, where the prover shows that it is storing *incompressible* data demonstrating the allocation of a lower-bound amount of resources.

Proofs of Space-time (PoSt) proposed by Moran and Orlov [15], is in a sense PoSpace over time. However, [15] only guarantees the dedication of space resources, not the stored data retrievability. In other words, the server only stores a randomly-generated string with no external utility to guarantee space dedication. The Filecoin project [16] introduced a PoSt scheme, where the server stores real data that can be used outside the protocol. Due to this important shift resource-wasting PoW schemes were replaced by a useful storage service. In [16], the prover executes sequential auditings, where each challenge is deterministically derived from the proof at a previous iteration and an input from a trusted randomness beacon. The prover chains the sequential challenges and proofs and compresses this chain using zk-SNARK, to be publicly inspected by the verifier. However, zk-SNARK is a heavy cryptographic machinery [4] entailing expensive computational/memory costs on the prover side, thus detering storage providers from renting storage to clients. Ateniese *et al.* [2] constructed a compact PoSt scheme based on TDF to obviate the need for zk-SNARKs.

3 Preliminaries

3.1 Merkle Tree and Merkle Mountain Range

A Merkle tree MT is a balanced binary tree with $n = 2^i$ leaves, such that every leaf holds the hash of a data block and every inner node is labelled with a hash of its children [14]. The inclusion of any data block in the tree can be proved only with a number of hashes logarithmic in the number of leaf nodes. A Merkle Mountain Range MMR [20], is a variant of Merkle tree that can be seen either as a list of perfectly balanced binary trees or a single binary tree truncated from the top right. Specifically, a MMR with root r is defined as a tree with $n = 2^i + j$ leaves, such that $i = \lfloor \log_2(n-1) \rfloor$. The left sub-tree $r.left$ can be seen as a MT with 2^i leaves, and the right sub-tree $r.right$ as a MMR with j leaves.

3.2 Proofs of Sequential Work

Proofs of Sequential Work (PoSW) first introduced by Mahmoody et al.[13] is a protocol between a prover P and a verifier V, where P can generate a proof convincing V that some computation took place for N time steps, since some

statement χ was received. The protocol is defined by algorithms PoSW, Open, and Verify as described below. P and V commonly input security parameters $w, c \in \mathbb{N}$ and a time parameter $N \in \mathbb{N}$. All parties have access to a random oracle $\mathsf{H} : \{0,1\}^* \to \{0,1\}^w$.

- PoSW: V samples a random statement $\chi \leftarrow \{0,1\}^w$ and sends it to P. P makes N sequential queries to H and computes a proof $(\phi, \phi_P) := \mathsf{PoSW}^{\mathsf{H}}(\chi, N)$, where ϕ is sent to V and ϕ_P is stored locally.
- Open: P computes $\tau := \mathsf{Open}^{\mathsf{H}}(\chi, N, \phi_P, \gamma)$ in response to random challenge $\gamma \leftarrow \{0,1\}^{c \cdot w}$ sampled by V. τ is then forwarded to V. The challenge can be generated non-interactively by the prover using the Fiat-Shamir [10].
- Verify: V outputs $\mathsf{Verify}^{\mathsf{H}}(N, \phi, \gamma, \tau) \in \{accept, reject\}$.

For a prover P and a verifier V honestly following the protocol's specifications, a complete PoSW protocol will output *accept* with probability 1. Soundness requires that even in the case of resourceful adversaries with parallel processing ability, a malicious prover cannot output a valid proof in time less than N. Cohen and Pietrzak [8] propose a simple PoSW construction based on a Merkle tree variant with added edges that connect the left siblings of a leaf's path to the root with the leaf itself in order to compute a leaf label. This graph is used for both sequential work enforcement and commitment purposes. In this paper, we use the terms PoET and PoSW interchangeably, since a PoSW protocol proves that N time has *elapsed* after χ was received.

3.3 Proof of Retrievability

Proof of Retrievability (PoR) schemes [11, 19] are a Proof of Storage category of protocols where a prover simultaneously ensures both possession and retrievability of a data file. PoR schemes consist of four algorithms (KeyGen, Store, Prove, Verify):

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$: Inputs λ and outputs secret/public key pair $(\mathsf{sk}, \mathsf{pk})$.
- $\mathsf{Store}(F^*, \mathsf{sk}) \to (F, \mathsf{tg})$: Takes original data file F^*, secret key sk, and generates encoded file F. It also outputs tag tg as necessary information to run $\mathsf{PoR.Prove}$ and $\mathsf{PoR.Verify}$ algorithms.
- $\mathsf{Prove}(\mathsf{pk}, chal, \mathsf{tg}, F) \to \pi$: Inputs file F, tag tg, public key pk, and challenge $chal$ issued by a verifier, and outputs proof π corresponding to the $chal$.
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}, \mathsf{tg}, chal, \pi) \to \{accept, reject\}$: Inputs secret/public key pair $(\mathsf{sk}, \mathsf{pk})$, tag tg, $chal$, proof π. Outputs a bit b to designate *accept* or *reject*.

The completeness property of a PoR scheme ensures that the protocol outputs *accept* with a probability of 1 for a prover and verifier honestly following the protocol's specifications. Loosely speaking, soundness requires an extractor algorithm that will recover the data through interaction with any prover that can pass the verification with overwhelming probability [19]. In other words, for any adversary $\mathcal{A}$ generating a valid proof π in the PoR protocol, there is an extractor algorithm $\mathsf{PoR.Ext}_{\mathcal{A}}(\mathsf{pk}, \mathsf{sk}, \mathsf{tg}; r)$ having as input pk, sk, the file tag tg, and

the description r of $\mathcal{A}$ (the random coin of $\mathcal{A}$), outputs the file F. PoR schemes also satisfy the *unpredictability* property ensuring that the prover cannot guess a valid response before it sees the corresponding challenge.

4 stoRNA Design

Now, we present our stoRNA, a stateless transparent PoSt protocol.

Ingredients and notation. stoRNA uses the following primitives:

–Collision-resistant hash function H with the output range of size w.

–Publicly verifiable $\mathsf{PoEt} = (\mathsf{Prove}, \mathsf{Verify})$ in [8] as a proxy for time (non-interactive version).

–Publicly verifiable stateless $\mathsf{PoR} = (\mathsf{KeyGen}, \mathsf{Store}, \mathsf{Prove}, \mathsf{Verify})$ scheme.

We denote concatenation of bit-strings by $\|$. For $x \in \{0,1\}^*$, $x[i \dots j]$ and $|x|$ denote concatenation of all bits from i^{th} bit to j^{th}, and bit-length of x, respectively.

Given PoR.Store algorithm outputs (processed file F and tag tg), a random seed rs, deposit time D, and audit frequency parameter T, stoRNA output is proof π that ensures a public verifier: (i) F is continuously available over time D, (ii) the prover did not learn the stoRNA output until D time after receiving F. The measure of time here is the number of sequential CPU hash invocations. We prove the following theorem.

Theorem 1. *Let* PoR *be a stateless PoR scheme with ϵ-soundness and unpredictability. Let* PoEt *be a PoEt scheme with δ-evaluation time. The time cost of* PoR *and hash function evaluation are negligible w.r.t. T. The time cost of s_0 sequential steps on the server processor is T'. If $T' + 2\delta D < T$, the proposed* PoSt *scheme (Algorithm 1) is stateless, complete, and ϵ-sound.*

4.1 Construction

The stoRNA scheme described in Algorithm 1 formally consists of three algorithms: stoRNA.Store, stoRNA.Prove, and stoRNA.Verify. In stoRNA.Store algorithm, the client only performs PoR.Store on an erasure encoded file F^* and outputs processed file F and tag tg to a prover. In the stoRNA.Prove algorithm, for every T time unit, the PoEt.Prove state will serve as the PoR challenge to generate a fresh PoR in PoR.Prove. Next, in order to commit to the whole chain of sequential $(\mathsf{PoEt}, \mathsf{PoR})$ proofs, the prover efficiently updates a graph as illustrated in Figure 2 and Algorithm 2.

This graph is based on the special DAG introduced in [8] with a number of modifications. Let $G_n^{\mathsf{Com}} = (V, E)$ be the commitment DAG, where each node in V is indexed by a bit string with a length at most n, while the root node is indexed by the empty string ϵ. Also, let $E = E' \cup E''$, where sub-graph (V, E') is a complete Merkle tree of depth n, with edges directed from the leaves coming up to the root. Index of each node in depth $i < n$ of the tree is made up of

the common bits of its parents. E.g., two parents indexed by $u = v \| 0$ and $u = v \| 1$ form a child indexed by v (Algorithm 2, line 5). Moreover, for all leaves $v \in \{0,1\}^n$, E'' consists of an edge (u, v) for any u that is a left node sibling on the path from v to the root ϵ (Algorithm 2, line 6).

Algorithm 1 stoRNA Construction

1: $\underline{\mathsf{stoRNA.Store}}$
2: input data file F^* and $(\mathsf{PoR.sk}, \mathsf{PoR.pk})$
3: $(F, \mathsf{tg}) \leftarrow \mathsf{PoR.Store}(\mathsf{PoR.sk}, \mathsf{PoR.pk}, F^*)$
4: sample random seed $rs \leftarrow\!\!\$ \{0,1\}^w$
5: output (rs, F, tg)
6: $\underline{\mathsf{stoRNA.Prove}}$
7: input processed file F, tag tg, random seed rs, deposit time D, and audit frequency T
8: set $i \leftarrow 0$ and $et \leftarrow 0$ ▷ i:number of audit iterations, et: elapsed time
9: set $\mathsf{st} \leftarrow rs$
10: **while** $et \le D$ **do**
11: $\quad \mathsf{st} \leftarrow \mathsf{PoEt.Prove}(T, \mathsf{st})$
12: $\quad i \leftarrow i + 1$
13: $\quad h_i \leftarrow \mathsf{st}$
14: $\quad c_i \leftarrow \mathsf{H}(h_i)$
15: $\quad \pi_i \leftarrow \mathsf{PoR.Prove}(\mathsf{PoR.pk}, F, \mathsf{tg}, c_i)$
16: $\quad l_\epsilon \leftarrow G_n^{\mathsf{Com}}.\mathsf{Update}(v = i, \mathcal{V} = h_i \| \pi_i)$ ▷ Algorithm 2
17: $\quad \mathsf{st} \leftarrow \mathsf{H}(\mathsf{st} \| l_\epsilon)$ ▷ update the state by appending the new G_n^{Com} root
18: $\quad et \leftarrow et + T$
19: $N \leftarrow i$
20: $H_{FNL} \leftarrow \mathsf{st}$
21: output $\mathsf{Com} = (H_{FNL}, l_\epsilon)$
22: $\underline{\mathsf{stoRNA.Verify}}$
23: input commitment Com, tag tg, seed rs, public key $\mathsf{PoR.pk}$
24: generate random $c-$element subset $I^* \subset [1, N]$ and send it to the prover.
25: wait to receive $\pi = \{h_i, \pi_i, \{l_k\}_{k \in \Delta_i}\}_{i \in I^*}$, where $\Delta_i = \{i[1, j-1] \| 1 - i[j]\}_{j \in [1,n]}$ ▷ Δ_i contains the index of all siblings of the nodes on the path from leaf i to the root as in Merkle tree commitment opening
26: **for all** $i \in I^*$ **do**
27: $\quad c_i \leftarrow \mathsf{H}(h_i)$
28: $\quad$ **if** $\mathsf{PoR.Verify}(\mathsf{PoR.pk}, \mathsf{tg}, \pi_i, c_i) = false$ **then** return false
29: $\quad$ **if** $\mathsf{PoEt.Verify}(T, h_i) = false$ **then** return false
30: $\quad$ **if** $l_i \neq \mathsf{H}(i, \pi_i, l_{p_1}, \ldots, l_{p_d})$, where $(p_1, \ldots, p_d) = \mathsf{Parents}(i)$ **then** return false
31: $\quad$ **if** $\exists j \in \Delta_i : l_j \neq \mathsf{H}(j, l_{j\|0}, l_{j\|1})$ **then** return false ▷ verify G_n^{Com} opening
32: return true

At iteration i, the prover updates the graph G_n^{Com} similarly to a Merkle mountain range described in subsection 3.1, also including additional E'' edges as described above. Besides, the label of node i is updated by appending the hash of the most

Algorithm 2 G_n^{Com}.Update

1: input index $v \in \{0,1\}^n$ and value $\mathcal{V}$, and DAG $G_n^{\mathsf{Com}} = (V, E)$, where $E = E' \cup E''$, sub-graph (V, E') is a Merkle tree, and E'' contains, for all leaves $v \in \{0,1\}^n$ an edge (u, v) for any u that is a left sibling of node on the path from v to the root ϵ.
2: $\mathsf{node_{count}} \leftarrow G_n^{\mathsf{Com}}.\mathsf{GetNodeCount}$ ▷ get total number of graph nodes
3: **if** $v > \mathsf{node_{count}}$ **then**
4: $\quad V \leftarrow V \cup v$ ▷ add leaf v to the tree
5: $\quad E' \leftarrow E' \cup \{(x \parallel b, x) : b \in 0, 1, |x| < n\}$ ▷ update Merkle tree edges starting from new leaf $v = x \parallel b$ to the root
6: $\quad E'' \leftarrow E'' \cup \{(i, v) : v = a \parallel 1 \parallel a', i = a \parallel 0\}$
7: $\quad l_v = \mathsf{H}(\mathcal{V}) \parallel \mathsf{H}(v, l_{p_1}, \dots, l_{p_d})$, where $(p_1, \dots, p_d) = \mathsf{Parents}(v)$
8: $\quad \forall i \in V, |i| < n : l_i = \mathsf{H}(i, l_{p_1}, l_{p_2})$, where $(p_1, p_2) = \mathsf{Parents}(i)$ ▷ recursively update all labels up to root
9: **else**
10: $\quad$ go to lines 7-8 to update labels
11: output l_ϵ

recent $(\mathsf{PoEt}, \mathsf{PoR})$ proofs to the labels of parents of node i. After an update to the commitment graph, the new root label l_ϵ is mixed into the state for the next PoEt execution. At the end of the deposit period, storNA.Prove algorithm outputs the latest root label l_ϵ together with the final PoEt state as a commitment to the chain of proofs sequentially generated during the entire period.

Upon receiving the commitment, in storNA.Verify algorithm, (i) the verifier challenges a randomly sampled subset of time slots, (ii) for every challenged time slot, the prover provides a Merkle opening together with all the $(\mathsf{PoEt}, \mathsf{PoR})$ proofs on the path from this challenged node to the root, (iii) the verifier, uses the commitment to check whether the returned proofs are located at the correct positions of the chain, and (iv) runs $\mathsf{PoEt.Verify}, \mathsf{PoR.Verify}$ algorithms to respectively verify the returned $\mathsf{PoEt}, \mathsf{PoR}$ proofs.

High level of security proof. For the soundness property, we need to prove that the largest time between two PoRs is less than T. Thus, for an honest prover P, any successive configurations of any time slot with a T length must contain at least a PoR proof. Then, following the soundness definition of PoR in 3.3, one can use the PoR extractor to recover the data from the partial configurations and the transition function. To recover the sequence of each computation epoch and feed it to an extractor, we use programability of random oracle. To this aim, we force PoSt provers inevitably query the random oracle, the challenge (except the first one) and response (except the last one) for each PoR via querying the random oracle H. Thus, the extractor can invoke a PoR extractor to extract the data by controlling H. We note that, our soundness proof exploits unpredictability of the random oracle[7]. Finally, we argue about the sequenciality of the scheme that follows the proof of sequentiality of Cohen and Pietrzak [8]. A malicious prover

[7] The unpredictability of the random oracle is important in the malicious prover case, as it is hard to let the extractor access each PoR's challenge and response

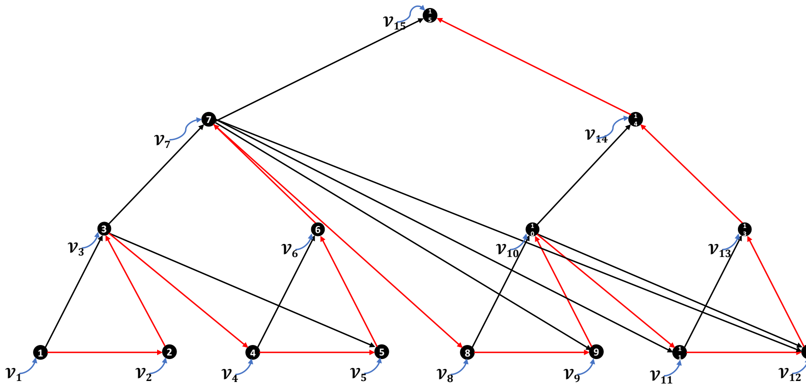


Fig. 2: A complete G_3^{Com} achieved after $N = 15$ iterations. Red lines show the traversing order of the tree with node numbers from 1 to $N = 2^{n+1} - 1$ for a tree of depth n and node i updated at iteration i. Also, $\mathcal{V}_i = (\mathsf{PoEt}, \mathsf{PoR})$ (Algorithm 1, line 16) shown in blue is input to $G_n^{\mathsf{Com}}.\mathsf{Update}$ algorithm at iteration i. (Color figure online)

$\mathsf{PoSt.P}'$, making the verifier accept (in relation to G_n^{Com} in Algorithm 2) with high probability must have queried H "almost" N times sequentially. We use the outputs of $\mathsf{PoR.Prove}$ as the input nodes of the specified tree construction of [8]. We defer the proof of Theorem 1 to appendix A.

5 Efficiency Analysis and Experimental Results

Implementation and experimental setup. We implement a prototype of the prover and the verifier in Golang[8]. Our testbed consisted of a MacBook Pro with 16 GB 3.22 GHz memory and a 2.06 GHz Intel Core i10 CPU with M1 (ARM based) chipset and Mac OS monterey as operating system. We implemented the scheme of [19] as our underlying stateless publicly verifiable PoR for randomly generated files of different sizes and relied on SHA-256 for all hash implementations. Following what presented in [8], here T and D are measured as the amount of sequential CPU steps. We refer to [21] for discussions on how it translates to real-world time. The results were averaged over 10 runs.

Setup Cost. stoRNA computation for the client solely includes running the $\mathsf{PoR.Store}$ algorithm once, no matter how long the deposit length D is. This cost is ignorable as compared with the setup cost of [2] which equals $1 \cdot \mathsf{PoR.Store} + N \cdot (\mathsf{TDF.TrapEval} + \mathsf{PoR.Prove})$, and $N = \frac{D}{T}$ denotes number of iterations during the deposit period D. As an example, the setup algorithm of [2] for a file of size 256 MB, stored for 5 months and checked on a 1-hour basis, takes about 200 minutes on a client machine. This time is prolonged for larger files or longer

[8] Code will be open-sourced soon and is available upon request.

deposit lengths. Our storNA.Store algorithm can be accomplished in a constant time $1 \cdot$ PoR.Store, independent of the deposit length.

Prover Cost. storNA.Prove algorithm makes a total of N sequential queries to PoEt, which is an intrinsically sequential process with overall steps proportional to the deposit length D. In mstoRNA.Prove, the average computational complexity per prover algorithm regarding PoEt computations is divided by m, assuming m as the number of Storage Nodes connected to a Time Node. Therefore, as m increases, the overall computational complexity of prover algorithm diminishes.

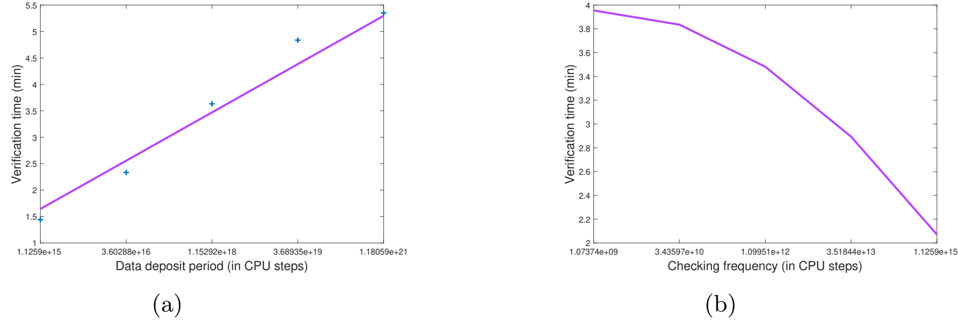
Verifier Cost. We now evaluate how our scheme verification time changes as the deposit period D grows. We fix the audit frequency parameter T to 2^{40} and vary the deposit period from 2^{50} to 2^{70} CPU steps. Since the results on various file sizes was roughly the same, we report the mean over all data files of sizes 64 MB, 128 MB, and 256 MB, with 10 experiments each. Figure 3a shows the results. As deposit length increases by $2^{20}\times$, the verification time grows from 1.64 minutes to 5.29 minutes, an increase of only $3\times$. This is because the number of nodes in each of c openings that the verifier algorithm checks their consistency is equal to the depth of the commitment graph, which grows logarithmically with the deposit length. We also explore how the change in audit frequency parameter T affects the verification time. For this experiment, we fix the the deposit period D to 2^{60} vary T from 2^{30} to 2^{50} CPU steps. For each configuration, we run 10 tournaments and measure the average of the verification time. Figure 3b shows the results. When $T = 2^{50}$, the verification time reaches the lowest, at 2.07 minutes. We also note that the verification algorithm is parallelizable, where nodes can be checked concurrently using verifier CUDA cores. In this prototype we have not implemented such parallelism and the results reflect the whole verification time without parallelism. mstoRNA construction shown in appendix B further optimizes the overall verification cost in the sense that PoEt sequence is inspected once for all m Storage Nodes connected to a Time Node.

Proof Size. The proof consists of the $\mathsf{Com} = (H_{FNL}, l_\epsilon)$ and c openings, each including n tuples of the form $\{h_k, \mathsf{PoEt}_k, \pi_k, l_k\}_{k \in \Delta_i}$, where $\Delta_i = \{i[1, j-1] \,\|\, 1 - i[j]\}_{j \in [1,n]}$. Table 2 report the results on proof sizes when varying deposit period D and audit frequency parameter T. With $w = 256$ bits, $c = 150$, which guarantees 2^{-50} security, $t = 39$ (i.e., over 10^{12} steps), and $n = 9$ (1024 total iterations), the proof size is approximately 1.7MB.

Discussion. Our construction is slower to verify and has larger proofs than the compact solution in [2]. This is the cost we pay for stateless and transparent features. Nonetheless, our *end-to-end* setup, proof, and verification costs are smaller than [2]. In Table 1, we give a high-level comparison with compact PoSt [2]. On top of potential parallelism possible in the verification mentioned earlier, there are potential ways of further optimizing performance: In addition to full-node verifiers inspecting the entire PoSt sequence, light client verification approaches like [12] are possible in storNA. Concretely, by adding intermediate "checkpoints" during the PoSt sequence computation, where each checkpoint includes the hash of the previous, a light client can verify directly through con-

D	t, n	Proof Size (MB)
2^{50}	29, 19	2.7930
	34,14	2.3940
	39,9	1.7550
2^{60}	29,29	4.2630
	34,24	4.1040
	39,19	3.7050
	44,14	3.0660
	49,9	2.1870
2^{70}	29,39	5.7330
	34,34	5.8140
	39,29	5.6550
	44,24	5.2560
	49,19	4.6170

Table 2: Proof sizes at various $t = \log \frac{T}{2}$ and $n = \log \frac{N}{2}$, with $N = \frac{D}{T}$. We assume $w = 256$ bits and $c = 150$, which guarantees 2^{-50} security.



(a) (b)

Fig. 3: storRNA.Verify algorithm time cost for $c = 150$. Solid lines show the trend. (a) Verification time when varying the deposit period D and audit frequency parameter $T = 2^{40}$. The overall verification cost is the same for all file sizes and logarithmic in the deposit length. (b) Verification time when varying audit frequency parameter T and deposit period $D = 2^{60}$.

secutive checkpoints and skip the validation of every time slot in PoSt sequence. Therefore, it is possible to audit data availability only for a specific time and not for the whole chain (*point verification*).

Acknowledgments. This work was funded by the HARPOCRATES EU research project (No. 101069535) and the Technology Innovation Institute (TII), UAE, for the project ARROWSMITH. Giulio Malavolta was partially funded by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972.

6 Conclusion

This work contributes towards building stateless transparent proofs of storage-time systems, whose design guarantees continuous availability of outsourced

data, and eventually have a tangible impact on building highly reliable storage services. The stateless, trapdoor-free, and dynamic nature of our construction together with the ignorable setup cost and adjustable prover difficulty properties render it applicable to modern real-world applications like decentralized storage networks and Blockchain history expiry where one can safely prune some of the on-chain data and have strong guarantees that the data will still be continuously available somewhere.

References

1. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 598–609. ACM (2007)
2. Ateniese, G., Chen, L., Etemad, M., Tang, Q.: Proof of storage-time: Efficiently checking continuous data availability. In: NDSS (2020)
3. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proceedings of the 4th international conference on Security and privacy in communication netowrks. ACM (2008)
4. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von neumann architecture. In: 23rd USENIX Security (2014)
5. Bertrand Portier: Always on: Business considerations for continuous availability. http://www.redbooks.ibm.com/redpapers/pdfs/redp5090.pdf, 2014
6. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Annual international cryptology conference. pp. 757–788. Springer (2018)
7. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. Journal of Cryptology **30**(1), 22–57 (2017)
8. Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 451–467. Springer (2018)
9. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: Annual Cryptology Conference. Springer (2015)
10. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
11. Juels, A., Kaliski Jr, B.S.: Pors: Proofs of retrievability for large files. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 584–597. ACM (2007)
12. Light Clients and Proof of Stake: https://blog.ethereum.org/2015/01/10/light-clients-proof-stake/
13. Mahmoody, M., Moran, T., Vadhan, S.: Publicly verifiable proofs of sequential work. In: Proceedings of the 4th conference on Innovations in Theoretical Computer Science. pp. 373–388 (2013)
14. Merkle, R.C.: Protocols for public key cryptosystems. In: Security and Privacy, 1980 IEEE Symposium on. pp. 122–122. IEEE (1980)
15. Moran, T., Orlov, I.: Simple proofs of space-time and rational proofs of storage. In: Annual International Cryptology Conference. pp. 381–409. Springer (2019)
16. Protocol Labs: Filecoin: A decentralized storage network (2018)

17. Rabaninejad, R., Attari, M.A., Asaar, M.R., Aref, M.R.: A lightweight auditing service for shared data with secure user revocation in cloud storage. IEEE Transactions on Services Computing **15**(1), 1–15 (2019)
18. Rabaninejad, R., Liu, B., Michalas, A.: Port: Non-interactive continuous availability proof of replicated storage. Cryptology ePrint Archive (2022)
19. Shacham, H., Waters, B.: Compact proofs of retrievability. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 90–107. Springer (2008)
20. Todd, P.: Merkle mountain range. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md
21. Wesolowski, B.: Efficient verifiable delay functions. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (2019)
22. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)

A Theorem 1 Proof

(i: Completeness): Directly follows from the completeness of the PoR and PoEt schemes.

(ii: Soundness): Let an adversary $\mathcal{A}$ be against the soundness of the stoRNA scheme. Let the extractor $\mathsf{PoSt.Ext} = (\mathsf{Ext}_{\mathsf{PoSt},1}, \mathsf{Ext}_{\mathsf{PoSt},2})$ recover the data F from the prover. Where $\mathsf{Ext}_{\mathsf{PoSt},1}$ on input the description of the prover, outputs the configurations, the epoch (a randomly chosen time slot) and the transition function, and $\mathsf{Ext}_{\mathsf{PoSt},2}$ is a PoR extractor that recovers the data from the configurations, the epoch and the transition function. Intuitively, we first show that the prover executes "one PoR" in a randomly chosen epoch and then by invoking the PoR extractor, we recover the data from the configurations the epoch and the transition function (extraction phase).

We first argue about the sequenciality in Algorithm 1. A potentially malicious prover $\mathsf{PoSt.P}'$, making the verifier accept (in relation to G_n^{Com} in Algorithm 2) with high probability must have queried H "almost" N times sequentially. The proof of sequentiality follows Cohen and Pietrzak [8]. We use the outputs of $\mathsf{PoR.Prove}$ as the input nodes of the specified tree construction of [8]. Thus, the sequentiality proof of Algorithm 1 follows the sequentiality proof of [8]. Formally we have that,

Lemma 1. [Theorem 1 of [8]]. Consider the scheme in Algorithm 1, with parameters c, w, N and a "soundness gap" $\alpha > 0$. If $\mathsf{PoSt.P}'$ makes at most $(1-\alpha)N$ sequential queries to H, and at most q queries in total, then $\mathsf{PoSt.V}$ will output reject with probability $1-(1-\alpha)^c - (2 \cdot n \cdot w \cdot q^2)/2^w$.

Where N is assumed to be number of sequential steps of the form $N = 2^{n+1} - 1$ for an integer $n \in \mathsf{N}$, and c is a statistical security parameter (the size of the subset I^* in which the larger the c the better the soundness), and w is the output range of H, which we need to be collision-resistant and sequential. $w = 256$ is a typical value. The proof follows the proof of Theorem 1 in [8].

In general, the verification algorithm of the stoRNA requires the prover to compute all PoR challenges and responses and evaluate the PoEts. Thus the PoR responses are valid and the PoEt are evaluated as expected with probability $(1-\alpha)^c - (2 \cdot n \cdot w \cdot q^2)/2^w$ based on Lemma 1. Because of the unpredictability of PoR and the sequentiality of PoEt, the PoR proofs must be generated sequentially.

Let D_0 and D_k be the start and end time points for running $\mathcal{A}$. For i from 1 to $k-1$, we set each time point D_{i+1} to be the first time when $\mathcal{A}$ queries the random

oracle H on $(\mathsf{st} \parallel l_\epsilon)$ (Alg.1 step 17). Similarly, we set each time point $\hat{D}_i$ as the start when $\mathcal{A}$ queries the H on st (Alg.1 step 13). Then we prove that the random time epoch with length $T > T' + 2\delta D$ chosen by $\mathsf{Ext}_{\mathsf{PoSt},1}$ must contain at least one interval $[D_i, \hat{D}_i)$ for some i. To this aim, we prove the following lemmas 2, 3, 4, and 5:

Lemma 2. The time point D_i must precede D_{i+1}.

Proof. we show that each PoEt's output st_{i-1} must be firstly queried to the random oracle H before st_i. To prove it we use the contradiction in a way that, if not, then $\mathcal{A}$ must be able to either generate the PoEt output st_i before st_{i-1}, which violates the sequentiality of PoEt; or generate the PoEt input st_{i-1} before the output of H (step 17, Algorithm 1), which violates the unpredictability of the random oracle H; or generate the PoR challenge c_i before st_{i-1}, which violates the unpredictability of the random oracle H (step 13, Algorithm 1); or generate the PoR response π_i before c_i, which violates the unpredictability of PoR;

Lemma 3. T' is shorter than the length of each time slot $[D_i, D_{i+1})$.

Proof. By the unpredictability of the random oracle, the output of the PoEt, st_i must be generated before the time point D_{i+1}. On the other hand, the PoR response π_i must be generated via the PoR on the challenge st_i after the time point D_i. Thus, a PoEt function must be evaluated within the time slot $[D_i, D_{i+1})$. By the sequentiality of PoEt, the length of $[D_i, D_{i+1})$ must be longer than T'.

Lemma 4. $T' + \delta D$ is bigger than the length of each time slot $[D_i, D_{i+1})$.

Proof. Let D' be the execution time of $\mathsf{PoSt.P}'$. By the correctness of the verification algorithm, $D' < (1 + \delta)D$. Based on the result of Lemma 3 , we have that the length of each time slot $[D_i, D_{i+1})$ is longer than T', thus, the longest slot should be shorter than $(1 + \delta)D - (k - 1)T' = \delta D + T'$.

Lemma 5. Each $\hat{D}_i \in [D_i, D_{i+1})$ and the time slot $[D_i, \hat{D}_i)$ is shorter than δD.

Proof. Finally, we show the PoEt response st_i must be queried to the random oracle H (Alg.1 step 13) within this time slot $[D_i, D_{i+1})$ and that the time slot $[D_i, \hat{D}_i)$ is shorter than δT. The output of the PoR π_i is queried at the time point D_{i+1}, hence the input of the PoR, c_i must be generated by $\mathsf{PoSt.P}'$ before the time D_{i+1} according to the sequentiality of PoR. Due to the unpredictability of the random oracle, H must be queried on input st_i before the time D_{i+1}. On the other hand, according to the unpredictability of PoEt, $\mathsf{PoSt.P}'$ can not figure out the PoEt proof st_i before the time point D_i, when the PoEt input is generated. Given this, st_i must be queried to the random oracle H in time slot $[D_i, D_{i+1})$. Furthermore, since the maximum length of $[D_i, D_{i+1})$ and the evaluation time of PoEt is longer than T', the slot $[D_i, \hat{D}_i) < \delta D$.

Extraction phase. In this phase, we show that given the bunch of configurations for $\mathsf{PoSt.P}'$ for time slot $[D_i, \hat{D}_i)$ (or $[D_{i-1}, \hat{D}_{i-1})$) and the code of the transition function, c_i and st_i can be accessed by the PoSt.Ext. Indeed, since both random oracles H are maintained by the extractor, a cheating prover of $\mathsf{PoR.P}'$ can be constructed by manipulating the output of the random oracle H (step 13, Algorithm 1) as the PoR challenge, rewinding the part of the $\mathsf{PoSt.P}'$ corresponding to time segment $[D_i, \hat{D}_i)$ and collecting the queries of the random oracle H (step 17, Algorithm 1) as the PoR response. Since there is a PoR extractor to recover the storage data from $\mathsf{PoR.P}'$, the soundness proof of PoSt is complete.

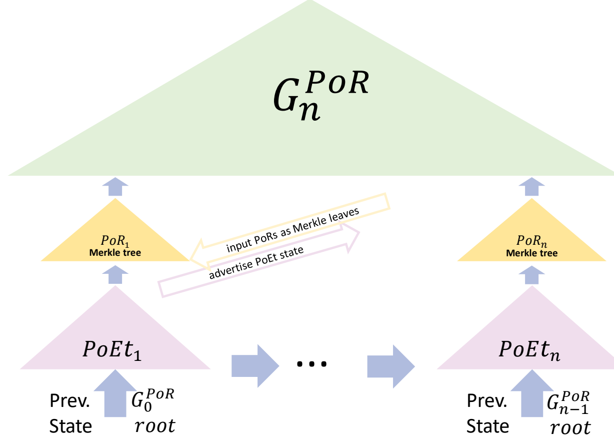


Fig. 4: Structure of the multi-prover stoRNA.

B Stateless Multi-Prover PoSt construction

In this section we show improvement options to the concrete efficiency of prover algorithm by proposing an extended multi-prover PoSt construction mstoRNA = (Store, Prove, Verify) (see Algorithm 3 for details). More precisely, we assume any arbitrary number of "Time Nodes" and "Storage Nodes" can freely join the DSN by respectively providing "CPU work" and "storage-time" resources to the network.

mstoRNA.Store algorithm is executed to output file F_j and tag tg_j which are outsourced to Storage Node j. In stoRNA.Prove algorithm, (i) Time Node, every T time units, shares the PoEt state and waits for a time gap determined by network latency. (ii) Storage Node j hosting file F_j, generates a challenge based on the freshly advertised PoEt state, serving as the PoR challenge, and submits $\pi_{ij} \leftarrow$ PoR.Prove. (iii) Time Node collects all PoR proofs from all Storage Nodes and creates a Merkle tree $\mathsf{MT_i}$ with root r_i. (iv) Time Node inputs r_i together with PoEt proof to update the commitment graph G_n^{Com}, and (v) Time Node timestamps the updated G_n^{Com} root, l_ϵ, into the shared PoEt state for the next PoEt execution. At the end of the deposit period D, the Time Node returns l_ϵ together with the final PoEt state as a commitment to all proofs sequentially generated during D.

Upon receiving the commitment, in mstoRNA.Verify algorithm, (i) the verifier challenges a randomly sampled subset of time slots (ii) for every challenged time slot, the Time Node provides openings for both G_n^{Com} and Merkle tree $\mathsf{MT_i}$ together with all the (PoEt, PoR) proofs on the path from this challenged node to the root, (iii) the verifier, verifies commitment openings of both $\mathsf{MT_i}$ and G_n^{Com}, and (iv) runs PoEt.Verify, PoR.Verify algorithms to respectively verify the returned PoEt, PoR proofs. As the number of Storage Nodes connected to a Time Node increases, the overall computational complexity of the prover algorithm diminishes. This enables even personal resource-constrained devices to partake in DSNs by dedicating some amount of disk-space, resulting in more decentralization. Besides, a PoSt sequence generated by a Time Node in mstoRNA can migrate to any other Time Node, who can continue where the previous prover left off. This is particularly important considering real nodes susceptible to Failure.

Algorithm 3 mstoRNA Construction

1: mstoRNA.Store
2: input data file F^* and (PoR.sk, PoR.pk)
3: $(F, \mathsf{tg}) \leftarrow \mathsf{PoR.Store}(\mathsf{PoR.sk}, \mathsf{PoR.pk}, F^*)$ ▷ repeat this for different files outsourced to m storage nodes
4: sample random seed $rs \leftarrow_{\$} \{0,1\}^w$
5: output (rs, F, tg)
6:
7: mstoRNA.Prove
8: **Time Node**
9: input random seed rs, deposit time D, and audit frequency T
10: set $i \leftarrow 0$ and $et \leftarrow 0$
11: set $\mathsf{st} \leftarrow rs$
12: **while** $et \leq D$ **do**
13: $\quad \mathsf{st} \leftarrow \mathsf{PoEt.Prove}(T, \mathsf{st})$
14: $\quad$ advertise st
15: $\quad i \leftarrow i + 1$
16: $\quad h_i \leftarrow \mathsf{st}$
17: $\quad$ wait to receive PoR proofs from storage nodes ▷ wait time is determined based on average network roundtrip time (RTT)
18: $\quad$ **for all** $j \in [1, m]$ **do**
19: $\quad\quad r_i \leftarrow \mathsf{MT.AppendLeaf}(\pi_{ij})$ ▷ create a Merkle tree with PoRs received from Storage Nodes
20: $\quad l_\epsilon \leftarrow G_n^{\mathsf{Com}}.\mathsf{Update}(v = i, \mathcal{V} = h_i \parallel r_i)$ ▷ Algorithm 2
21: $\quad \mathsf{st} \leftarrow \mathsf{H}(\mathsf{st} \parallel l_\epsilon)$
22: $\quad et \leftarrow et + T$
23: $N \leftarrow i$
24: $H_{FNL} \leftarrow \mathsf{st}$
25: output $\mathsf{Com} = (H_{FNL}, l_\epsilon)$
26:
27: **Storage Node**
28: node j input processed file F_j, tag tg_j, deposit time D, and audit frequency T
29: periodically input advertised state st
30: $c_i \leftarrow \mathsf{H}(\mathsf{st})$
31: $\pi_{ij} \leftarrow \mathsf{PoR.Prove}(\mathsf{PoR.pk}, F_j, \mathsf{tg}_j, c_i)$
32: output π_{ij}
33:
34: mstoRNA.Verify
35: input commitment Com, tag $\{\mathsf{tg}_j\}_{j\in[1,m]}$, seed rs, public key PoR.pk
36: generate random $c-$element subset $I^* \subset [1, N]$ and send it to all provers.
37: wait to receive $\{h_i, r_i, \pi_{ij}, \{l_k\}_{k\in\Delta_i}\}_{i\in I^*}$, where $\Delta_i = \{i[1, j-1] \parallel 1 - i[j]\}_{j\in[1,n]}$
38: ▷ Δ_i contains commitment openings for both G_n^{Com} and $\mathsf{MT_i}$
39: **for all** $i \in I^*$ **do**
40: $\quad c_i \leftarrow \mathsf{H}(h_i)$
41: $\quad$ **for all** $j \in [1, m]$ **do**
42: $\quad\quad$ **if** $\mathsf{MT.Verify}(r_i, \pi_{ij}) = \mathit{false}$ **then** return false ▷ verify $\mathsf{MT_i}$ opening
43: $\quad\quad$ **if** $\mathsf{PoR.Verify}(\mathsf{PoR.pk}, \mathsf{tg}_j, \pi_{ij}, c_i) = \mathit{false}$ **then** return false
44: $\quad$ **if** $l_i \neq \mathsf{H}(i, r_i, l_{p_1}, \ldots, l_{p_d})$, where $(p_1, \ldots, p_d) = \mathsf{Parents}(i)$ **then** return false
45: $\quad$ **if** $\exists j \in \Delta_i : l_j \neq \mathsf{H}(j, l_{j\parallel 0}, l_{j\parallel 1})$ **then** return false ▷ verify G_n^{Com} opening
46: $\quad$ **if** $\mathsf{PoEt.Verify}(T, h_i) = \mathit{false}$ **then** return false
47: return true

Theorem 2. *Let* PoR *be a stateless PoR scheme with ϵ-soundness and unpredictability. Let* PoEt *be a PoEt scheme with δ-evaluation time. The time cost of* PoR *and hash function evaluation are negligible w.r.t. T. The time cost of s_0 sequential steps on the server processor is T'. If $T' + 2\delta D < T$, the proposed* mstoRNA *scheme in Algorithm 3 is stateless, complete, and ϵ-sound.*

Proof. (i: Completeness): This property directly follows from the completeness of the PoR and PoEt schemes.

(ii: Soundness): The proof strategy of the mstoRNA scheme directly follows the proof in Theorem 1. The mstoRNA verification algorithm requires the provers to compute PoR challenges (and responses) in the Storage Node phase, and evaluate the PoEt in the Time Node phase as in Algorithm 3. Therefore, based on Lemma 1, with the probability $(1-\alpha)^c - (2 \cdot n \cdot w \cdot q^2)/2^w$, one can conclude that the PoR responses are valid and the PoEt are evaluated as expected, due to sequentially of PoEt and unpredictability of PoR. For *Extraction phase*, the random time slot with a length longer than T must contain at least one PoR execution, and both the input and output of PoR can be located via random oracle H. This makes the extraction works in a way that one can invoke PoR extractor to recover the data.