PoRt: Non-Interactive Continuous Availability Proof of Replicated Storage

Reyhaneh Rabaninejad
Tampere University, Finland
reyhaneh.rabbaninejad@tuni.fi

Bin Liu
Tampere University, Finland
zkMe Labs
bin.liu@tuni.fi

Antonis Michalas
Tampere University, Finland and
RISE Research Institutes of Sweden
antonios.michalas@tuni.fi

ABSTRACT

Secure cryptographic storage is one of the most important issues that both businesses and end-users take into account before moving their data to either centralized clouds or blockchain-based decentralized storage marketplace. Recent work [4] formalizes the notion of Proof of Storage-Time (PoSt) which enables storage servers to demonstrate non-interactive continuous availability of outsourced data in a publicly verifiable way. The work also proposes a stateful compact PoSt construction, while leaving the stateless and transparent PoSt with support for proof of replication as an open problem. In this paper, we consider this problem by constructing a proof system that enables servers to simultaneously demonstrate *continuous availability* and *dedication of unique storage resources* for encoded replicas of a data file in a *stateless* and publicly verifiable way. We first formalize Proof of Replication-Time (PoRt) by extending PoSt formal definition and security model to provide support for replications. Then, we provide a concrete instantiation of PoRt by designing a lightweight replica encoding algorithm where replicas' failures are efficiently located through an efficient *comparison-based* verification process, after the data deposit period ends. PoRt's proofs are *aggregatable*: the prover can take several sequentially generated proofs and efficiently aggregate them into a single, succinct proof. The protocol is also stateless in the sense that the client can efficiently extend the deposit period by incrementally updating the tags and without requiring to download the outsourced file replicas. We also demonstrate feasible extensions of PoRt to support dynamic data updates, and be transparent to enable its direct use in decentralized storage networks, a property not supported in previous proposals. Finally, PoRt's verification cost is independent of both outsourced file size and deposit length.

KEYWORDS

Data Outsourcing, Continuous Data Availability, Proof of Replication

ACM Reference Format:
Reyhaneh Rabaninejad, Bin Liu, and Antonis Michalas. 2023. PoRt: Non-Interactive Continuous Availability Proof of Replicated Storage. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555776.3577741

Corresponding author: Reyhaneh Rabaninejad.

SAC '23, March 27-31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577741

1 INTRODUCTION

Storage-as-a-Service, including cloud storage services and more recent Decentralized Storage Networks (DSNs) [17, 25], has attracted extensive interest and caused big data migration from local storage systems to the storage servers as it offers efficient and scalable services at a lower cost. However, the data owner has no physical control over the data after outsourcing. Hence data availability throughout deposit time is an important trait that highly reliable storage providers [8] 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) [3, 14] ensures data integrity and availability only at a specific time point (i.e. the time the challenge is issued).

To achieve *continuous availability* guarantee of outsourced data, Ateniese *et al.* [4] formalized the notion of Proof of Storage-time (PoSt), which provides tools for storage servers to convince a verifier that the server has dedicated storage space over a specified time interval. A naive PoSt scheme using PoS with frequent checks over time, however requires the clients to be online when sending sequential challenges over time to the server. Moreover, in DSNs such as Filecoin [17], where proofs are verified by the blockchain network, this method causes communication complexities and, potentially, leads to network bottlenecks. The authors in [4] propose a compact PoSt construction based on Trapdoor Delay Functions (TDF). The idea is that client executes a pre-processing phase to generate a tag, where he performs the same work as the prover, but with faster TDF evaluations due to his knowledge of the trapdoor. The data file together with an initial challenge are uploaded to a server at the beginning of the deposit period and the client is no longer required to be online. During the deposit period, the server generates chained challenge-proof pairs, where each challenge is the TDF output of the previous proof, and *hashes* all the challenge-proof pairs to generate a compact proof. In the verification phase, only the equality of this hash is checked with the pre-processed tag. Although the idea of hashing all the challenge-proof pairs makes the scheme compact and fast to verify, but it also leads to being *stateful* (the number of audit interactions between the prover and verifier is bounded) and *static* (data cannot be updated after outsourcing). Besides, when the number of audits reaches the a priori bound (deposit period ends), the client to extend the deposit period requires to *download the entire data* to generate new tags. Furthermore, the

soundness of the 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. The authors have pointed several aspects that remain unresolved such as: (i) support for data replication, (ii) statelessness and support for dynamic data updates, (iii) transparent (without trapdoor) PoSt constructions, and (iv) setup cost reduction.

Data replication is typically mentioned as a guarantee for continuous data availability in Service Level Agreements (SLAs) [1]. It is formally captured by the notion of Proof-of-Replication (PoRep), which guarantees that a file is fully replicated onto different servers– thus ensuring data availability in case of failures, power outages, or even attacks on the servers. Similar to PoS, existing PoRep proposals [7, 13] are interactive, requiring the client's involvement to repeatedly send fresh challenges over time in the auditing process.

In light of these issues we ask the following question: *Is it possible to construct an efficient, non-interactive, yet stateless mechanism for continuous availability monitoring of replicated data at storage providers?*

1.1 Contributions

This work makes significant progress in answering the above question. In particular, we construct a proof system that enables service providers to simultaneously demonstrate *continuous availability* and *dedication of unique storage resources* for encoded replicas of a data file over a deposit period, in a stateless and publicly verifiable manner. Our contributions can be summarized as follows:

- **C1. Proof of Replication-Time Formalization**: We introduce the notion of Proof of Replication-Time (PoRt). Our framework tackles continuous availability proof of replicated storage. To the best of our knowledge, no prior proposal for Proof of Replication-Time exists.
- **C2. PoRt Construction:** We design a candidate construction of PoRt with properties of public verifiability, compactness, statelessness, and usefulness. The idea is to replicate the content via proposed linear encoding algorithm. The client then pre-processes encoded replicas to generate verification tags, with the dominant computation executed only once for all encoded replicas. As a result, the proposed construction is scalable with regards to replication factor as oppose to the **naive** approach which is using a distinct PoSt execution per individual file replica.
- **C3. PoRt Features:** The proofs/tags in the proposed construction are *aggregatable*: the prover/client can respectively take several sequentially generated proofs/tags and efficiently aggregate them into a single, succinct proof/tag. This aggregatable feature also makes tags and proofs much more convenient to update, making the protocol incremental: prover/client can keep up the proof/tag sequence from the last state to aggregate further proofs/tags. The protocol is also *stateless* in the sense that the client can efficiently extend the deposit period by incrementally updating the tags and without requiring to download the outsourced file replicas. After deposit period ends, replica failures are detected in an efficient verification algorithm by using the idea of *comparing* proofs on distinct encoded replicas, instead of verifying all storage proofs one-by-one. The scheme is also proven secure under the proposed definition. We further show the feasibility of enhancing the proposed construction to be *dynamic* and *transparent* by discussing ideas at the end of the paper.
- **C4. Efficiency:** From the performance point of view, PoRt's computations are *independent of the file size*, which in turn result in its scalability with regards to large files. The verification cost is also independent of both the file size and deposit length. Furthermore, PoRt's succinct proofs enable automatic public verification via on-chain smart contracts.

1.2 Technical Overview

Consider a client wishing to replicate and disperse its data to the storage provider(s) and verify continuous availability of all replicas *at once with a single protocol run* and without remaining online during the whole storage period. Such a goal can be achieved with the use of PoRt. It proceeds as follows: The client performs Store = (rEncode, TagGen) algorithm to (i) generate l encoded file replicas in an efficient lightweight rEncode algorithm and (ii) generate replica tags as necessary information for the prover and public verifier in TagGen algorithm. Each storage provider, participating in the PoRt protocol, stores some of the file replicas for a specific deposit period. To prove "Replication-time" i.e. continuous availability of the specified replica over the specified deposit time, the storage provider frequently generates storage proofs for each file replica during the entire storage period in Prove algorithm. To compel a specific amount of delay between successive storage proofs generated by the prover, the protocol leverages RSA Time-Lock Puzzle (TLP). We note that this chain of TLPs is generated only once for all file replicas at both server and client sides, which makes the protocol easy to scale for any replication factor. All sequential proofs generated on each encoded replica over deposit period are aggregated to generate a succinct proof.

After deposit period ends, the public verifier runs the Verify algorithm to locate replica failures. Here, we introduce an elegant technique: to audit continuous availability, instead of verifying each and every one of storage proofs generated during deposit period which incurs huge cost on the public verifier, we use the replicated nature of the data in the protocol and the *homomorphic* property of rEncode algorithm to first *unmask* the replica coefficients from the proofs and then *compare* the unmasked proofs to audit file replicas and locate the corrupted ones. The only assumption we make here is that majority of replicas are intact and available during the deposit period so that the corrupted ones are detected in this comparison process. Figure 1 depicts a schematic overview of our designed PoRt construction.

The above construction, seems to provide the following desirable features: (i) *Public verifiability*, (ii) *Compactness*: The verification cost is independent of the file size and deposit length. (iii) *Statelessness and unbounded use*: For example, consider the case where the client initially stores his files for three months, and then decides to extend the contract for another month. PoRt enables the client to easily extend the deposit period without requiring to download the outsourced file replicas. This is possible due to the *incremental*

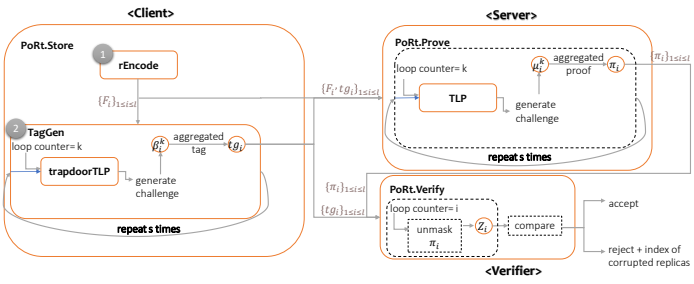


Figure 1: Designed construction of PoRt**.**

nature of the protocol: Both the client and the provers can respectively keep up the tag and proof sequence from the last state to aggregate further tags/proofs. Since the TagGen procedure only depends on the replication coefficients and not the data file, this extending process does not require the client to download outsourced data file from the servers. The client, in order to update the tags for extended deposit period, only needs to efficiently re-compute replication coefficients as in rEncode algorithm and continues the TagGen procedure from the last state to generate updated tags for an extended deposit period. Moreover, since the verification cost is independent of the deposit length, this deposit time extension *does not affect* the cost of verification algorithm. In Appendix A, we will discuss this statelessness feature of the protocol and will provide some ideas on how PoRt can be extended to also support *dynamic data operations*, and be *transparent*.

2 RELATED WORK

Proofs of Storage (PoS) schemes enable clients to outsource file 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 which shows that the server is still intactly storing the client's file. 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) [5]. Proofs of Retrievability (PoR) schemes [14] have similar concept to PDP, but they also guarantee data retrievability, which is achieved by an extractor that reconstructs the client's file from the proofs returned by the prover. An extensive research exists on PDP/PoR schemes covering various features including dynamic data updates [10, 18, 19], and shared data files [20, 21].

Proofs of Replication (PoRep) schemes are a type of PoS protocols where instead of proving file possession, the storage server should prove it possesses multiple replicas of the same file. Therefore, PoRep enables the prover to ensure the verifier that each independent replica of some file has been dedicated an independent physical storage. Different categories of proofs of replication such as private/public replication and private/public verification exist. Private replication occurs in the instances where the client generates the encoded replicas on his own in a pre-processing phase, while public replication refers to the case where the complete replication process is outsourced to a server. All these categories are explored in the literature – e.g. private replication-private verification [2, 11], private replication-public verification [12], and public replication-public verification [17, 23]. Public replication is achieved by using no secrets in file replication (transparent property, as defined by [17]). However, to resist the *generation attack*, i.e. preventing the server from generating a file replica on-the-fly at the time of generating a proof, Lerner [23] used time-asymmetric encodings to *slow down* the replication process. Filecoin [17] introduced *Seal*, slowable pseudo random permutations constructed by using a block cipher in cipher block chaining mode made publicly verifiable using SNARKs [6]. The slow property of replication makes it almost impossible for an adversary to generate replicas just-in-time when responding to a challenge.

Proofs of Space-Time (PoSt) The notion of PoSt proposed by Moran and Orlov [16], is in a sense PoS over time, i.e. it proves a dedication of space resources over a period of time. However, [16] only guarantees dedication of space resources, not retrievability of the data stored in that disk-space. In other words, the server only stores a randomly-generated string with no external utility to guarantee space dedication. A PoSt scheme in which the server stores real data that can be used outside of the protocol, was introduced in the Filecoin project [17]. This was an important shift as it enabled the replacement of the resource-wasting PoW with a useful storage service. In [17], the prover executes sequential auditings where each challenge is deterministically derived from the proof at previous iteration. The prover chains the sequential challenges and proofs and compresses this chain using SNARKs [6] together with a proof of the elapsed time, to be inspected all at once by the verifier. However, SNARK is a very heavy cryptographic machinery which incur expensive computational/memory costs at the prover side, economically disincentivizing storage providers in renting storage to clients. In a recent work, Ateniese *et al.* [4] constructed a compact PoSt scheme based on the idea of sequential proofs chain, which employs a trapdoor delay function (TDF) to obviate the need for SNARKs. Instead of verifying chained challenge-proof pairs, the verifier pre-computes same chained TDFs as the prover in pre-processing phase, but without delay due to his knowledge of the trapdoor. Later in the verification phase, only the equality of the pre-processed hashed chain is checked with the final compact proof. Notwithstanding the progress, several aspects including support for data replication and being stateless are left as open problem.

3 PRELIMINARIES

3.1 Proof of Replication

Proof of Replication enables a prover to ensure the verifier that each independent replica of some file is dedicated an independent physical storage. A Proof of Replication scheme consists tuple of algorithms PoRep = (Setup, rEncode, Prove, Verify):

- $\mathsf{Setup}(1^\lambda) \rightarrow (param, \mathsf{sk})$: Takes as input a security parameter λ and outputs the public parameters $param$ and a secret key sk.
- $\mathsf{rEncode}(F, l, \mathsf{sk}) \rightarrow (\{F_i\}_{1 \le i \le l}, tg)$: Takes as input an original data file F, the replication factor l, and secret key sk, and generates l distinct encoded replicas $\{F_i\}_{1 \le i \le l}$. It also outputs a tag tg as necessary information to run PoRep.Prove and PoRep.Verify algorithms.

- Prove($param, chal, tg, \{F_i\}_{1\le i\le l}) \rightarrow \pi$: Takes as input replicas $\{F_i\}_{1\le i\le l}$ for a file F, tag tg, the public parameters $param$, a random challenge $chal$ issued by a verifier and outputs π – a storage proof for replica F_i.
- Verify($param, tg, chal, \pi) \rightarrow \{0, 1\}$: Takes as input the public parameters $param$, tag tg, the challenge $chal$ and a proof π. It outputs a bit b – $b = 1$ if the verifier accepts the proof, $b = 0$ otherwise.

3.2 RSA Trapdoor Time-Lock Puzzle

This construction proposed by Rivest *et al.* [22] is an inherently sequential repeated exponentiation in a group of unknown order based on the hardness of RSA factoring. The $\mathsf{TLP}(x, t)$ for an RSA modulus $N = p \cdot q$ is slowly evaluated as $y = x^{2^t} \in \mathbb{Z}_N$ by performing 2^t iterated squarings. There is no parallel algorithm that can perform t squarings modulo N significantly faster than just performing t squarings sequentially. However, the verifying party who knows Euler trapdoor function $\phi(N)$, can reduce the exponent to $t' = 2^t \mod \phi(N)$, and thus efficiently compute $y = x^{t'} \mod N$. More precisely, the protocol is defined by algorithms Gen, TLP, and trapdoorTLP as described below.

- $\mathsf{Gen}(1^\lambda) \rightarrow N$: Samples two safe primes $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are also primes and output $N = p \cdot q$.
- $\mathsf{TLP}(1^\lambda, N, x) \rightarrow y$: Takes as input $x \in \mathbb{Z}_N$, and evaluates $y = x^{2^t} \in \mathbb{Z}_N$ by performing 2^t iterated squarings.
- $\mathsf{trapdoorTLP}(1^\lambda, N, x, \phi(N)) \rightarrow y$: Takes as input $x \in \mathbb{Z}_N$ and Euler trapdoor function $\phi(N)$, and computes $t' = 2^t \mod \phi(N)$, and $y = x^{t'} \mod N$.

The RSA time-lock puzzle is also called the RSW problem after Rivest, Shamir and Wagner who first proposed it. The following definition of RSW problem is adapted from [15].

Definition 3.1 (t-RSW problem). Let Gen be the generation algorithm of the RSA time-lock puzzle. The t-RSW problem is (t_p, t_o, ϵ)-hard if for every polynomial-time adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ where $\mathcal{A}_1$ runs in time t_p and $\mathcal{A}_2$ runs in time t_o such that $t_o < t$, it holds that:

$$\Pr\left[y = x^{2^t} \;\middle|\; \begin{array}{l}(N, p, q) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda), state \leftarrow\!\$\ \mathcal{A}_1(1^\lambda, N),\\ x \leftarrow\!\$\ \{2, \ldots, N-1\}, y \leftarrow\!\$\ \mathcal{A}_2(1^\lambda, x, state)\end{array}\right] \le \epsilon(\lambda).$$

4 FORMALIZING PROOF OF REPLICATION-TIME

PoRt helps the servers to convince the verifier that each encoded replica of same data file is available throughout a period of time at the server side. More specifically, PoRt involves three main entities: client, servers and verifier. The client is the data owner who wishes to replicate and outsource its data to storage service providers. It only involves in the setup stage for generating encoded replicas, public parameters and the an challenge for the servers. The public parameters need to be available among the entities while the encoded replicas and the initial challenge will be distributed to the servers via authenticated channels. After that, the client does not need to be online since no further interaction is required between it and the other entities. The servers are contracted with storing the individual replicas and generating the corresponding proofs. The verifier is tasked with the verification of the proofs generated by the servers with the use of the public parameters regarding the data storage task. The verifier does not necessarily need to be the client itself if the scheme allows for public verification.

Here, we follow similar style with [4] to consider time parameters in defining PoRt syntax. Consider T as the time period a specific data file is supposed to be deposited in the server. T is divided into time slots of length t, where t is the audit frequency parameter. This helps approximating continuous data availability throughout time range T with discretized frequent auditing, where smaller t provides a superior availability guarantee. The measure of time here is the number of unit steps of the Turing machine. We now formalize PoRt which consists of a tuple of four algorithms PoRt = (Setup, Store, Prove, Verify), based on sequential use of PoReps introduced in subsection 3.1:

- $\mathsf{Setup}(1^\lambda, t, T) \rightarrow (param, sk)$: Inputs security parameter 1^λ, audit frequency parameter t, and deposit time T and outputs the public parameters $param$ and secret key sk.
- $\mathsf{Store}(F, l, sk, t, T) \rightarrow (\{F_i\}_{1\le i\le l}, tg)$: Takes original data file F, replication factor l, secret key sk, audit frequency parameter t, and deposit time T and generates l distinct file replicas $\{F_i\}_{1\le i\le l}$ by running $\mathsf{PoRep.rEncode}(F, l, sk)$. A global $timer$ will be initialized to allow the verifier to check if the proofs are received on time. It also outputs a tag tg as necessary information to run PoRt.Prove and PoRt.Verify algorithms.
- $\mathsf{Prove}(param, chal, tg, \{F_i\}_{1\le i\le l}) \rightarrow \pi$: Inputs file replicas $\{F_i\}_{1\le i\le l}$, tag tg, public parameters $param$, and random challenge $chal$ issued by a verifier, and outputs π as the storage proof for replica F_i at least once in every time slot t. These proofs can be aggregated into a single compact proof which is verified at the end of deposit period T.
- $\mathsf{Verify}(param, sk, tg, chal, \pi, timer) \rightarrow \{0, 1\}$: Inputs $param$, secret key sk, tag tg, challenge $chal$, proof π, and $timer$ to check whether the final proof is received on time. It outputs a bit $b = 1$ if the verifier accepts the proof and $b = 0$ otherwise.

The following, are desirable design features of a PoRt scheme.

Public Verifiability. To allow any third party to verify continuous data availability without downloading data from the servers. To this end, the verification algorithm PoRt.Verify should not take secret key sk as input.
Compactness. To enable low overhead verification with cost independent of the file size and deposit length.
Usefulness. To provide storage system for storing real data rather than storing a randomly-generated string with no external use.
Stateless. To support polynomial unbounded number of verifications without requiring the verifier to maintain protocol state.
Dynamic. To enable clients efficiently update outsourced data at any time without the need to an entirely new setup.
Transparency. A PoRt scheme may import a one-time trusted setup run by an honest client where the setup output is publicly published to all entities. However, a PoRt 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 any malicious client-prover to generate a valid proof at the time a challenge is

issued by generating data on-the-fly to collect network rewards, without really reserving storage.

In addition, a PoRt scheme must provide the following security properties.

Completeness. For a client and servers who honestly follow the protocol steps, the proof generated in PoRt.Prove algorithm during deposit period T by the honest servers hosting file replicas will be accepted in PoRt.Verify algorithm. More precisely, for all $(param, sk)$ values output by PoRt.Setup$(1^\lambda, t, T)$, all files $F \in \{0,1\}^*$, and all $(\{F_i\}_{1\leq i\leq l}, tg)$ output by PoRt.Store(F, l, sk, t, T), a proof π generated by prover in PoRt.Prove$(param, chal, tg, \{F_i\}_{1\leq i\leq l})$ on the challenge $chal$ will make PoRt.Verify$(param, sk, tg, chal, \pi, timer)$ always output 1.

Soundness. The soundness property of a PoRt scheme guarantees that a server is able to convince an honest verifier that it has stored independent replicas of a file throughout the specified deposit time only if it actually has.

To capture this security requirement, we adopt the security definition of soundness for PoSt schemes introduced in [4] and extend it to support for replications. In general, we require that if a server that act as the prover can pass the verification procedure that involves an honest verifier with non-negligible probability, then there exists an extractor that can extract the replicas during any period specified by the audit frequency and also at the last moment when the storage order is completed with overwhelming probability. Likewise, a prover here is modelled as an Interactive Turing Machine (ITM) which facilitates data extraction during some specific time range (measured in Turing Machine steps) and therefore allows for verifying continuous data possession.

More specifically, we define soundness for PoRt schemes via the following game between an adversary $\mathcal{A}$ and a challenger that acts as an honest verifier. The challenger first runs PoRt.Setup to generate the public parameter $param$ and the secret key sk and then hands $param$ to the adversary. After that, $\mathcal{A}$ needs to specify a challenge on some file F by calling the Challenge oracle. On receiving the query, it runs the PoRt.Store algorithm to generate l encoded replicas $\{F_i\}_{1\leq i\leq l}$ and a tag tg for future verification. Challenge can only be called once and all further queries to it will be ignored.

The adversary can start the verification procedure by querying the Audit oracle with a tag tg'. If tg' is generated by a previous query to Challenge, Audit will then start the auditing procedure with the adversary. For simplicity, in the case that the public parameter $param$ and the secret key sk are clear in the context, we let the verification algorithm only take the tag as input. At the end of the auditing procedure, $\mathcal{A}$ will be provided a single bit b indicating that whether the verification successes or not. Even the adversary is engaged in any auditing procedure, it can still call Audit to initiate other instances of the auditing procedure with respect to the same tag at the meantime.

At some point during the game, $\mathcal{A}$ outputs a tag tg^* with the ITM description of a set of provers $\{\mathcal{P}_i\}_{1\leq i\leq l}$. We require that tg^* is generated by the query to Challenge and the auditing procedure between the honest verifier and the set of provers succeeds with probability at least ϵ (in such case, we call the set of provers ϵ-*admissible*). The auditing procedure is slightly different here: if a timer has been initialized in a previous run of the PoRt.Store algorithm which generates tg^* then it will be reinitialized again. For the set of provers provided by the adversary, if there exists an efficient extractor Extr that can successfully extract the corresponding replica, the experiment returns 1 and otherwise 0. The PoRt scheme is considered to be sound if for any efficient adversary that attacks against the scheme, the experiment returns 1 with overwhelming probability.

We further require that the extractor Extr works as follows. It takes as input the public parameter $param$, the secret key sk, a tag tg and the ITM descriptions of the provers $\{\mathcal{P}_i\}_{1\leq i\leq l}$ output by the adversary. For simplicity, when the other parameters are clear in the context, we just let the extractor take the ITMs as the input. Extr first selects a random index j from the range $\{1, ..., l\}$ and then runs instances of the provers. During the execution, Extr records each configuration yields *only* by $\mathcal{P}_j$ and also the elapsed time when executing each step. On the completion of the auditing procedure, Extr randomly selects t consecutive configurations from the record and then tries to extract the replica F_j from: (1) the transition function of the ITM; or (2) both the selected t' configurations and the configurations within the time interval $[T, T+\delta]$ for some δ specified by the verification procedure. It captures the intuition that the data should be available at any time period t during the deposit period T and also at the end of deposit time. Notice that valid proofs submitted within the time interval $[T, T+\delta]$ will be accepted, the replica should therefore be available at some point during this period in order to guarantee continuous data retrivability throughout the deposit time.

It is important to highlight three points here. First, our security definition of soundness aims to guarantee *continuous availability (more exactly, retrieveability) of encoded replicas*, but not requiring the replicas are stored strictly in somewhat intact form (e.g. the replicas cannot be compressed to some certain extent). Providing such a guarantee is an interesting open future work and our work can be considered as a step towards this goal. In other words, our security definition allows that the replicas being stored can be processed in arbitrary ways but it is ensured that all the individual replicas are retrievable at every "check point" specified by the audit frequency during the deposit time. Therefore, using a carefully chosen small audit frequency will provide better guarantee on data retrivability.

Second, we want to provide a general model for capturing soundness of PoRt schemes and therefore there is no explicit restriction on the form of the auditing procedure. It can be of the challenge-response form with arbitrary many interactions, stateless or stateful, and with the use of some timers or not, etc. Thus, it allows for capturing different features of PoRt schemes.

Third, the definition of soundness for PoSt schemes presented in [4] cannot thwart the attack that a malicious server discards the data after generating the last proof as required. Even though both of their constructions can ensure data availability till the last moment, their definition cannot provide such a guarantee. More specifically, a PoSt scheme that allows the prover to generate the last proof at some early point in the last time interval can be proven to be secure according their security definition. To rule out this attack, our security definition of soundness further requires that the extractor should be able to retrieve the replicas from both the

randomly selected configurations and also from the configurations at the end of data deposit period.

Definition 4.1 (Soundness). A PoRt scheme Π = (PoRt.Setup, PoRt.Store, PoRt.Prove, PoRt.Verify) for audit frequency parameter t, deposit time T and replication parameter l is said to be ϵ-sound if for any probabilistic polynomial-time ϵ-admissible adversary $\mathcal{A}$, there exists an extractor Extr that works as described above such that

$$\mathbf{Adv}^{\text{sound-(t,T,l)}}_{\Pi,\mathcal{A}}(\lambda) := 1 - \Pr[\mathbf{Exp}^{\text{sound-(t,T,l)}}_{\Pi,\mathcal{A}}(\lambda) \rightarrow 1] \leq \mathsf{negl}(\lambda),$$

where negl is a negligible function in λ and the experiment $\mathbf{Exp}^{\text{sound-(t,T,l)}}_{\Pi,\mathcal{A}}$ is defined as follows:

Experiment $\mathbf{Exp}^{\text{sound-(t,T,l)}}_{\Pi,\mathcal{A}}(\lambda)$

$chal \leftarrow \emptyset$
$(param, sk) \leftarrow\$ \text{PoRt.Setup}(1^\lambda, t, T)$
$(tg^*, \{\mathcal{P}_i\}_{1\leq i\leq l}) \leftarrow\$ \mathcal{A}(1^\lambda, param : O)$
If $tg^* \in chal \ \wedge F_j \leftarrow\$ \text{Extr}(\{\mathcal{P}_i\}_{1\leq i\leq l})$ then
  Return 1
Else return 0

Challenge(F)	Audit(tg')
If $chal \neq \emptyset$ then return $\perp$ $(\{F_i\}_{1\leq i\leq l}, tg) \leftarrow\$ \text{PoRt.Store}(F, l, sk, t, T)$ $chal \leftarrow chal \cup \{tg\}$ Return $(\{F_i\}_{1\leq i\leq l}, tg)$	If $tg' \notin chal$ then return $\perp$ $b \leftarrow\$ <\text{PoRt.Verify}(tg') \rightleftharpoons \mathcal{A}>$ Return b

Figure 2: The oracles O that $\mathcal{A}$ has access to in $\mathbf{Exp}^{\text{sound-(t,T,l)}}_{\Pi,\mathcal{A}}$.

5 PORT: A COMPACT PROOF OF REPLICATION-TIME CONSTRUCTION

In this section, we present our construction of PoRt that includes the tuple of four algorithms PoRt = (Setup, Store, Prove, Verify) described as follows:

– PoRt.Setup($1^\lambda, t, T$). This is run by the client and takes as input a security parameter 1^λ, the audit frequency parameter t, and the deposit time T. Let $N = p \cdot q$ be a publicly known RSA modulus, where $p = 2p' + 1$ and $q = 2q' + 1$ are two safe primes where p' and q' are also primes. All operations are performed in multiplicative cyclic group $\mathbb{Z}^*_N$ of invertible integers modulo N. The private order of $\mathbb{Z}^*_N$ is the Euler trapdoor function $\phi(N) = (p-1)\cdot(q-1)$. The tuple of algorithms SE = (KeyGen, Enc, Dec) denotes a semantically secure symmetric encryption scheme. $\mathcal{H}$, $\mathcal{H}'$, and $\mathcal{G}$ are secure hash functions. The client selects a pseudo-random permutation (PRP) $f : y \times \{0,1\}^{\log_2 n} \rightarrow \{0,1\}^{\log_2 n}$ and two pseudo-random functions (PRF) over domain D, $\psi : D \times \{0,1\}^{\log_2 l} \rightarrow D$ and $\theta : D \times \{0,1\}^{\log_2 n + \log_2 l} \rightarrow D$, with secret keys $k_1, k_2 \in D$, respectively. Parameter l is the data replication factor agreed in the SLA, and S is the number of squarings modulo N per unit of time that can be computed by the prover, determined based on a reasonable estimation of hardware speed of individual servers as in computational timestamping [9]. A data file F encoded with an erasure code, is divided into n blocks $m_i \in D$ denoted as $F = (m_1, \ldots, m_n)$. The public parameters are $param = (N, \mathcal{H}, \mathcal{H}', \mathcal{G}, f, l, S, t, T)$, and the secret key is $sk = (\phi(N), \psi, \theta, k_1, k_2)$.

– PoRt.Store($F, sk, param$). This is run by the client and has two subroutines rEncode and TagGen. It takes as input a data file $F = (m_1, \ldots, m_n)$, the secret key sk, the public parameters $param$. It constructs l distinct replicas of file F denoted as $\{F_i\}_{1\leq i\leq l}$ using replica encoding algorithm rEncode. It also runs the TagGen algorithm to generate tag tg. To this end, the client computes $e = tS$, chooses a random initial challenge $c^{(0)}$, and creates $s = \lceil T/t \rceil$ sequential puzzles with e squarings each. Note that each puzzle inputs the challenge outputted by previous puzzle. The client performs all sequential puzzles *without delay* using the trapdoor, and aggregates them into one tag tg_i used for verification of encoded replica F_i. We note that the dominant computation in TagGen algorithm which is solving puzzles is similar for all encoded replicas and thus executed only once. Finally, the client sends the encoded replicas $\{F_i\}_{1\leq i\leq l}$ and the challenge seed $c^{(0)}$ to the server and initiates global timer *timer* by resetting it to 0. The tag tg is also published publicly.

– PoRt.Prove($\{F_i\}_{1\leq i\leq l}, tg, param, c^{(0)}$). For simplicity, we present this algorithm for a single server's case, which means the server will store all the replicas on its side and then aggregate the sequentially generated proofs. For the case of multiple servers, the PoRt.Prove algorithm for each server can be easily derived from the one presented here. Upon receiving replicas $\{F_i\}_{1\leq i\leq l}$ together with tag tg, the public parameters $param$, and challenge $c^{(0)}$ the deposit period officially starts. Therefore, the prover starts sequential proof

PoRt.Store

Input data file $F = (m_1, \ldots, m_n)$, the secret key sk, and public parameters $param$

rEncode
Compute $\alpha_i = \psi_{k_1}(i)$
For $j = 1$ *to* n
  $\beta_{ij} \leftarrow \theta_{k_2}(i \parallel j)$
  $m_{ij} \leftarrow \alpha_i m_j + \beta_{ij}$
$F_i \leftarrow (m_{i1}, \ldots, m_{in})$
Return F_i //output i^{th} replica F_i

TagGen
Set $e \leftarrow tS$, $s = \lceil T/t \rceil$, and $(tg_1, \ldots, tg_l) \leftarrow (1, \ldots, 1)$
Sample $c^{(0)} \leftarrow\$ \mathbb{Z}^*_N$
Compute $e' \leftarrow 2^e \bmod \phi(N)$
Run SE.KeyGen algorithm to generate symmetric encryption key k
For $k = 1$ *to* s
  $a \leftarrow \mathcal{H}(c^{(k-1)})$
  $c^{(k)} \leftarrow a^{e'} \in \mathbb{Z}_N$ //sequentially solve puzzles without delay using trapdoor $\phi(N)$
  $r \leftarrow \mathcal{H}'(c^{(k)})$
  $J \leftarrow \emptyset$
  For $j \in [1, c]$ //use puzzle output $c^{(k)}$ to generate $J \subset [1, n]$ as a c-element subset of block indices to be challenged in auditing step k
    $J \leftarrow J \cup f_r(j)$
    $y^{(k)}_j \leftarrow \mathcal{G}(c^{(k)}, j)$
  For $i = 1$ *to* l //generate aggregated tag tg_i for replica F_i
    $\beta^{(k)}_i \leftarrow \sum_{j\in J} y^{(k)}_j \beta_{ij}$
    $tg_i \leftarrow tg_i \times (c^{(k)})^{-\beta^{(k)}_i} \bmod N$
$a \leftarrow \mathcal{H}(c^{(s)})$
$c^{(s+1)} \leftarrow a^{e'} \in \mathbb{Z}_N$
$K \leftarrow \mathsf{k} + c^{(s+1)} \bmod N$ //encrypt the key k using the time-release encryption
$\alpha \leftarrow \text{SE.Enc}_{\mathsf{k}}(\alpha_1, \ldots, \alpha_l)$ //encrypt $(\alpha_1, \ldots, \alpha_l)$ with symmetric encryption key k
$tg \leftarrow (K, \alpha, tg_1, \ldots, tg_l)$
Return tg

PoRt.Prove

Input replicas $\{F_i\}_{1\le i\le l}$, tag tg, public parameters $param$, and challenge $c^{(0)}$

Set $(\pi_1,\ldots,\pi_l) \leftarrow (1,\ldots,1)$
Parse tg as $(K, \alpha, tg_1,\ldots,tg_l)$
For $k = 1$ *to* s
  $a \leftarrow \mathcal{H}(c^{(k-1)})$
  $c^{(k)} \leftarrow a^{2^e} \in \mathbb{Z}_N$ //sequentially solve puzzles
  $r \leftarrow \mathcal{H}'(c^{(k)})$
  For $j \in [1, c]$ //use puzzle output $c^{(k)}$ to generate a fresh challenge
    $J \leftarrow J \cup f_r(j)$
    $y_j^{(k)} \leftarrow \mathcal{G}(c^{(k)}, j)$
  For $i = 1$ *to* l //generate aggregated proof π_i for replica F_i
    $\mu_i^{(k)} \leftarrow \sum_{j\in J} y_j^{(k)} m_{ij}$
    $\pi_i \leftarrow \pi_i \times (c^{(k)})^{\mu_i^{(k)}} \mod N$
$\pi \leftarrow (\pi_1,\ldots,\pi_l)$
Return π
$a \leftarrow \mathcal{H}(c^{(s)})$
$c^{(s+1)} \leftarrow a^{2^e} \in \mathbb{Z}_N$
$k \leftarrow K - c^{(s+1)} \mod N$ //retrieve key k using time-release decryption
$(\alpha_1,\ldots,\alpha_l) \leftarrow \mathsf{SE.Dec}_k(\alpha)$ //decrypt $(\alpha_1,\ldots,\alpha_l)$ using the obtained key k
Return $(k, \alpha_1,\ldots,\alpha_l)$

process to provide an on-time continuous availability guarantee of replicated storage. To this end, the prover runs PoRt.Prove to solve s sequential time-lock puzzles throughout time T, and aggregates them into one final proof π_i for each encoded replica F_i. The prover also outputs private vector $(\alpha_1,\ldots,\alpha_l)$ which can only be discovered sequentially after proof expiration time, thanks to the time-release encryption technique used in the scheme.

– PoRt.Verify$(param, tg, \pi, timer)$. Upon receiving proof π and vector $(\alpha_1,\ldots,\alpha_l)$ from the prover, verifier first checks if the $timer$ is in the interval $[T, T+\delta]$, where δ is a constant parameter determined based on time-lock puzzle evaluation. It also checks if $(\alpha_1,\ldots,\alpha_l)$ values are received at time interval $[T+t, T+\delta']$, with some $\delta' \ge t+\delta$. If these time constraints pass, the verifier then checks PoRt.Verify for a fast and efficient fault localization of corrupted replicas. If PoRt.Verify outputs reject and the value Z_i is not equal to other values, the corresponding replica is identified as corrupted.

PoRt.Verify

Input proof π, tag tg for a file F, and vector $(k, \alpha_1,\ldots,\alpha_l)$

Parse π as $(\pi_1,\ldots,\pi_l)$ and tg as $(K, \alpha, tg_1,\ldots,tg_l)$

Check if $\mathsf{SE.Dec}_k(\alpha) \stackrel{?}{=} (\alpha_1,\ldots,\alpha_l)$
For $i = 1$ *to* l
  $Z_i \leftarrow (\pi_i.tg_i)^{\frac{1}{\alpha_i}}$ //unmask π_i using tg_i and α_i
If $\exists i \in [1,l] : Z_i \ne majority(Z)$ return $reject$ //compare each unmasked value Z_i with the majority value in vector $Z = (Z_1,\ldots,Z_l)$ to locate the corrupted replicas
Return *accept*

6 SECURITY ANALYSIS

Having proposed the formal security definitions for PoRt schemes, in this section we will examine the security properties of the construction PoRt with respect to the security definitions we provided. We first show that *Completeness* is preserved in our construction.

Theorem 6.1. *If the client and server(s) are honest,* PoRt *preserves Completeness.*

Proof. Consider the value Z_i generated in PoRt.Verify:

$$\begin{aligned}
Z_i = (\pi_i.tg_i)^{\frac{1}{\alpha_i}} &= \Big(\prod_{k=1}^{s} (c^{(k)})^{\mu_i^{(k)}} \cdot \prod_{k=1}^{s} (c^{(k)})^{-\beta_i^{(k)}}\Big)^{\frac{1}{\alpha_i}} \\
&= \Big(\prod_{k=1}^{s} (c^{(k)})^{\sum_{j\in J} y_j^{(k)} m_{ij}} \cdot \prod_{k=1}^{s} (c^{(k)})^{-\beta_i^{(k)}}\Big)^{\frac{1}{\alpha_i}} \\
&= \Big(\prod_{k=1}^{s} (c^{(k)})^{\sum_{j\in J} y_j^{(k)} (\alpha_i m_j + \beta_{ij})} \cdot \prod_{k=1}^{s} (c^{(k)})^{-\beta_i^{(k)}}\Big)^{\frac{1}{\alpha_i}} \\
&= \Big(\prod_{k=1}^{s} \big((c^{(k)})^{\alpha_i \sum_{j\in J} y_j^{(k)} m_j + \sum_{j\in J} y_j^{(k)} \beta_{ij}} \cdot (c^{(k)})^{-\beta_i^{(k)}}\big)\Big)^{\frac{1}{\alpha_i}} \\
&= \Big(\prod_{k=1}^{s} \big((c^{(k)})^{\alpha_i \sum_{j\in J} y_j^{(k)} m_j + \beta_i^{(k)}} \cdot (c^{(k)})^{-\beta_i^{(k)}}\big)\Big)^{\frac{1}{\alpha_i}} \\
&= \Big(\prod_{k=1}^{s} (c^{(k)})^{\alpha_i \sum_{j\in J} y_j^{(k)} m_j}\Big)^{\frac{1}{\alpha_i}} = \prod_{k=1}^{s} (c^{(k)})^{\sum_{j\in J} y_j^{(k)} m_j}
\end{aligned}$$

Writing the same equalities for $Z_{i'}, i' \ne i$ also yields $\prod_{k=1}^{s} (c^{(k)})^{\sum_{j\in J} y_j^{(k)} m_j}$, where m_js are the original file blocks $F = (m_1,\ldots,m_n)$. This shows that if the proofs are generated based on intact encoded replicas, then Z_i and $Z_{i'}$ would be equal. Note that every single proof in sequential proof-chain during time T must be generated correctly; if any proof fails to verify, the whole Z_i fails to verify. □

The theorem for *soundness* and its proof are an important step towards achieving practical security for PoRt schemes. We leave as future work to improve on aspects such as computation model that deals with real-world time for defining security and more relaxed assumption on the security proof.

Theorem 6.2. *If ψ and θ are secure PRFs, f is a secure PRP,* SE *is IND-CPA secure and RSW problem is computationally hard,* PoRt *is ϵ-sound in the random oracle model under the assumption that the time cost of the computation other than solving the RSA time-lock puzzles is negligible.*

Proof Sketch. We prove the theorem by using a sequence of games which we describe below.

Game 0: The initial game is the original experiment that defines soundness of PoRt schemes in Definition 4.

Game 1: In this game, the experiment proceeds as the previous one with the following changes. We replace every $\psi_{k_1}(i)$ with a random value chosen uniformly from the domain D. The challenger who is in charge of the game keeps a record of those selected random values for further verification. If there exists a distinguisher which can distinguish the success probability of the adversary between the two games, it can be used to break security of the PRF ψ.

Game 2: This game proceeds the same as the one above, with the exception that we replace every $\theta_{k_2}(i)$ with a random value chosen uniformly from the domain D. Still, the challenger maintains a list to record the random values for further verification. Similarly, any distinguisher who can distinguish the success probability of the adversary between the two games can be used to break security of the PRF θ.

Game 3: We now transform *Game* 2 into *Game* 3 by replacing every invocation to the PRP f in Challenge with a function f'

which is chosen uniformly at random from the set of permutation mappings from $\log_2 n$-bits strings to $\log_2 n$-bit strings. If there is a distinguisher which can tell the difference in the adversary's success probability between *Game* 2 and *Game* 3, it can be used to break security of the pseudo-random permutation. Since in an execution of the algorithm PoRt.Store will compute f under s different keys, the reduction we obtain here will suffer a security loss by a factor of s.

Now we elaborate on the advantage that an adversary $\mathcal{A}$ can gain in *Game* 3. To win the game, the instances of the ITMs that $\mathcal{A}$ provides should succeed in the auditing procedure with probability at least ϵ. Therefore, we further distinguish between the following two situations in which $\mathcal{A}$ can win the game: (1) The data blocks stored by the server are not enough to generate valid proofs to pass the verification, or (2) The server has stored enough data blocks to generate valid proofs.

For the first case, we consider two possible situations based on the fact whether the data blocks can be retrieved by the servers when they are needed for computing the proofs. First, if the data blocks can be retrieved when needed, it means that the transition function has specified the method which allows the server to do so. Since we only consider *rational* adversaries, in such case the data blocks may be stored locally in various forms but the extractor should be able to retrieve the data blocks for reconstructing the replica by following the instructions specified by the transition function. Second, if the data blocks cannot be retrieved when needed for generating the proofs, the only possibility is that the adversary can gain advantage from the symmetric encryption scheme and therefore manage to forge proofs with α_i values in order to pass the verification check (i.e. by coming up with the proofs π_i, π_j and the values α_i, α_j such that $(\pi_i \cdot tg_i)^{\frac{1}{\alpha_i}} = (\pi_j \cdot tg_j)^{\frac{1}{\alpha_j}}$ holds for every $i, j \in \{1, \ldots, l\}$). Then we modify *Game* 3 and obtain the following game.

Game 4: In this game, the challenger does not encrypt the content $(\alpha_1, \ldots, \alpha_l)$ but a random string of the same length instead. The rest of the game remains the same as the previous one. Let E denote the event of situation (1) mentioned above. When E happens, the adversary does not need to obtain the key k via solving the last time-lock puzzle. Because it is required that the proofs need to be submitted before that then $\mathcal{A}$ needs to come up with the proofs and the corresponding $(\alpha_1, \ldots, \alpha_l)$ values at some earlier point. Therefore the difference in the success probability of the adversaries in *Game* 3 and *Game* 4 is bounded by the probability of the event E occurs. A distinguisher that aims to distinguish between the two games can be used to break indistinguishability under chosen-plaintext attack of the symmetric encryption scheme SE.

Finally, we discuss the success probability of the adversary in *Game* 4. Since the encrypted content of $(\alpha_1, \ldots, \alpha_l)$ is replaced by a random string, the adversary can win this game only under the situation (2) mentioned above. Notice that in *Game* 4, α_i and β_{ij} are uniform random numbers, the encoded data block m_{ij} are uniformly distributed and does not depend on any other data block. Moreover, the set J used in each round in the algorithms TagGen and PoRt.Prove contains a random subset of elements in $\{1, \ldots, n\}$ which cannot be determined in advance. The prover will also have to call the random oracles in turns so that the generated proofs can be consistent. So, if the server has stored enough data blocks required for generating the proofs and everything is computed correctly, an extractor can directly retrieve the data blocks from any consecutive configurations in which the execution time sums up to t and also the configurations for computing the last proof and then applies erasure code to reconstruct the replica. Therefore, the only possibility that $\mathcal{A}$ can win the game is via gaining some advantage from solving the RSA time-lock puzzles. In such case, the adversary $\mathcal{A}$ is assumed to solve *at least an instance* of the RSA time-lock puzzles by performing significantly less operations than the number of operations required to solve the puzzle when generating the proofs for the specified challenge. Therefore, given such an adversary, we can construct another adversary for solving tS-RSW problem that uses $\mathcal{A}$ as a subroutine by simulating to it the *Game* 4 with the challenge carefully planted in it, where t is the audit frequency and S is the number of modular squarings per unite of time estimated based on the server's hardware.

To sum up, assuming the PRFs, the PRP, the symmetric encryption scheme are secure and the RSW problem is computationally hard, the adversary's success probability in initial game which defines soundness of the PoRt scheme is bounded by a negligible function and therefore completes the proof of Theorem 6. □

7 PERFORMANCE EVALUATION

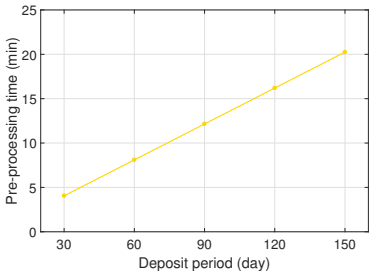
To evaluate the performance of PoRt, we ran experiments on a laptop with configuration Intel I7-3470 3.20 GHz processor, 4 GB memory and Windows 7 operating system using Crypto++ library Version 8.2. We relied on SHA-256 for all hash implementations. The RSA modulus used here was of 1024 bits size. We also set number of challenged blocks in each auditing to be $c = 460$, which guarantees 0.99 probability of detecting corruption according to [3]. The results were averaged over 10 runs.

Client Overhead. PoRt computation for the client includes encoding file replicas (rEncode) and generating tag (TagGen). For rEncode, $l \cdot (n + 1)$ PRF instantiations are required, where l and n denotes replication factor and total number of file blocks, respectively. Besides, TagGen includes s sequential trapdoor puzzle computations. Figure 3 shows experimental results for various deposit time T, frequency parameter t, replication factor l, and file sizes, respectively. As shown in Figure 3a, PoRt.Store algorithm for replication factor 2 , 30 days deposit period, and hourly checkup took 4.05 minutes. As shown in Figure 3d, this time remains almost fixed for different file sizes, due to the fact that rEncode cost is negligible compared to TagGen overhead.

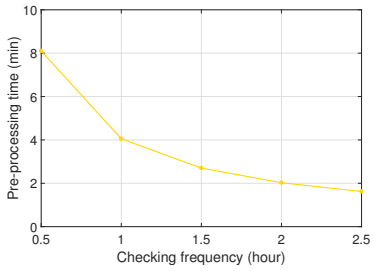
Prover Overhead. PoRt.Prove algorithm cost is dominated by computing time-lock puzzles, which is an intrinsically sequential process with total number of $T \cdot S$ iterated squarings for deposit period T and any replication factor l.

Verifier Overhead. PoRt.Verify algorithm performs l exponentiation and multiplications modulo N. E.g., for two replicas ($l = 2$), the smart contract verifying the proof roughly costs 80000 gas (the gas cost is 34.34 Gwei [1]), equivalent to 0.002605 ETH which is constant for all file sizes, deposit periods, and checking frequencies.

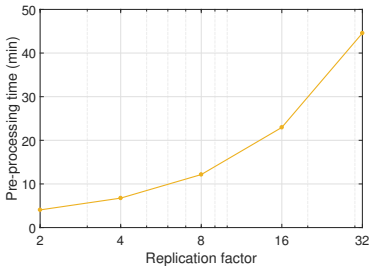
[1]The gas cost is the average gas price for October 21, 2022, as stated in https://ycharts.com/indicators/ethereum_average_gas_price



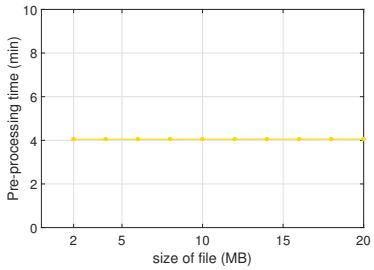
(a) different deposit periods, $l = 2$, and hourly checkup.



(b) different checking frequencies, $l = 2$, and 30 days deposit period.



(c) different replication factors, hourly checkup, and 30 days deposit period.



(d) different file sizes, $l = 2$, hourly checkup, and 30 days deposit period.

Figure 3: PoRt.Store **algorithm time cost for entire data file with** $c = 460$

Communication Overhead. In order to enable automatic public verification of proofs via smart contracts, the proofs should be succinct to reduce the expensive on-chain smart contract interactions. In PoRt, the proof size equals one RSA group element per each replica and thus has total length of $1024 \cdot l$ bits for l encoded replicas. E.g., for two replicas, the proof length is 256 bytes.

8 CONCLUSION

This work contributes towards building a compact proof system that will, by design, guarantee continuous availability of replicated storage, and eventually have a tangible impact on building highly reliable storage services. As an interesting future work, we consider making PoRt transparent to enable its direct use in Decentralized Storage Networks.

9 ACKNOWLEDGMENTS

This work was funded by Technology Innovation Institute (TII), UAE, for the project ARROWSMITH: Living (Securely) on the edge.

REFERENCES

[1] Amazon S3 Service Level Agreement. [n.d.]. https://aws.amazon.com/s3/sla/.
[2] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O Karame. 2016. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *25th USENIX Security Symposium (USENIX Security 16)*.
[3] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security, ACM 2007*. ACM, 598–609.
[4] Giuseppe Ateniese, Long Chen, Mohammad Etemad, and Qiang Tang. 2020. Proof of storage-time: Efficiently checking continuous data availability. NDSS.
[5] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. 2008. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*. ACM.
[6] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*. Springer, 90–108.
[7] Juan Benet, David Dalrymple, and Nicola Greco. 2017. Proof of replication. *Protocol Labs, July* 27 (2017), 20.
[8] Bertrand Portier. [n.d.]. Always on: Business considerations for continuous availability. http://www.redbooks.ibm.com/redpapers/pdfs/redp5090.pdf, 2014.
[9] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.
[10] David Cash, Alptekin Küpçü, and Daniel Wichs. 2017. Dynamic proofs of retrievability via oblivious RAM. *Journal of Cryptology* 30, 1 (2017), 22–57.
[11] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. 2008. MR-PDP: Multiple-replica provable data possession. In *2008 the 28th international conference on distributed computing systems*. IEEE, 411–420.
[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. 2019. Proofs of replicated storage without timing assumptions. In *Annual International Cryptology Conference*. Springer, 355–380.
[13] Ben Fisch. 2019. Tight proofs of space and replication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer.
[14] Ari Juels and Burton S Kaliski Jr. 2007. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*. Acm, 584–597.
[15] Jonathan Katz, Julian Loss, and Jiayu Xu. 2020. On the Security of Time-Lock Puzzles and Timed Commitments. In *Theory of Cryptography Conference*. Springer.
[16] Tal Moran and Ilan Orlov. 2019. Simple proofs of space-time and rational proofs of storage. In *Annual International Cryptology Conference*. Springer, 381–409.
[17] Protocol Labs. 2018. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf.
[18] Reyhaneh Rabaninejad, Mahmoud Ahmadian Attari, Maryam Rajabzadeh Asaar, and Mohammad Reza Aref. 2019. Comments on a lightweight cloud auditing scheme: Security analysis and improvement. *Journal of Network and Computer Applications* 139 (2019), 49–56.
[19] Reyhaneh Rabaninejad, Mahmoud Ahmadian Attari, Maryam Rajabzadeh Asaar, and Mohammad Reza Aref. 2020. A lightweight identity-based provable data possession supporting users' identity privacy and traceability. *Journal of Information Security and Applications* 51 (2020), 102454.
[20] Reyhaneh Rabaninejad, Mahmoud Ahmadian Attari, Maryam Rajabzadeh Asaar, and Mohammad Reza Aref. 2022. A Lightweight Auditing Service for Shared Data with Secure User Revocation in Cloud Storage. *IEEE Transactions on Services Computing* 15, 1 (2022), 1–15. https://doi.org/10.1109/TSC.2019.2919627
[21] Reyhaneh Rabaninejad, Seyyed Mahdi Sedaghat, Mohamoud Ahmadian Attari, and Mohammad Reza Aref. 2020. An ID-Based Privacy-Preserving Integrity Verification of Shared Data Over Untrusted Cloud. In *2020 25th International Computer Conference (CSICC)*. 1–6. https://doi.org/10.1109/CSICC49403.2020.9050098
[22] Ronald L Rivest, Adi Shamir, and David A Wagner. 1996. Time-lock puzzles and timed-release crypto. (1996).

[23] Sergio Demian Lerner. 2014. Proof of unique blockchain storage. https://bitslog.wordpress.com/2014/11/03/proof-of-local-blockchain-storage/.
[24] Benjamin Wesolowski. 2019. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 379–407.
[25] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.
[26] Yan Zhu, Huaixi Wang, Zexing Hu, Gail-Joon Ahn, Hongxin Hu, and Stephen S Yau. 2011. Dynamic audit services for integrity verification of outsourced storages in clouds. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM.

A EXTENSIONS AND FUTURE DIRECTIONS

The proposed PoRt scheme satisfies properties of public verifiability, compactness, statelessness, and usefulness. In this section, we show in a high-level how PoRt can be extended to support dynamic data operations, and be transparent. We hope that this section, apart from showing our future direction, can give valuable insights to researchers who wish to either enhance our work or focus on the design of similar and even better PoRt systems.

Statelessness. PoRt can be invoked a polynomial unbounded number of audits between the prover and verifier. When the number of verifications reaches the a priori bound (deposit period ends), the client can easily extend the deposit period without requiring to download the outsourced file replicas. This is possible due to the incremental nature of the protocol: Both the client and the prover can respectively keep up the tag and proof sequence from the last state to aggregate further tags/proofs at the agreed frequency. The client, in order to update the tags for extended deposit period, only needs to re-compute $\beta_{ij} = \theta_{k_2}(i \parallel j)$ values and continues the TagGen procedure from the last state to generate $\beta_i^{(k)}$ and update the tags to $(tg'_1, \ldots, tg'_l)$ for an extended deposit period T'. We note that since the TagGen procedure does not depend on the data file, this updating process does not require downloading outsourced data file from the servers.

However, one more challenge needs to be addressed here: Encrypting the vector $(\alpha_1, \ldots, \alpha_l)$ with time-release encryption does not work in this case, since the *time* extends by updating the tags. Not publishing vector $(\alpha_1, \ldots, \alpha_l)$ results in private verification. One approach to address this challenge is to send $(g^{\alpha}_1, \ldots, g^{\alpha}_l)$ publicly, and change the Z_is in PoRt.Verify to $Z_i = e(\pi_i.tg_i, g^{\frac{1}{\alpha_i}})$, where e is a bilinear pairing. This however, increases the verification cost.

Dynamic Data Operations. In PoRt, the coefficients $\beta_{ij} = \theta_{k_2}(i \parallel j)$ depend on the index j of block m_{ij}. Therefore, when updating an outsourced data file by inserting/deleting a block, all block indices and their β_{ij} coefficients are affected. Therefore, the tag requires to be entirely re-computated in the Setup algorithm. To enable clients to efficiently update data stored in the server at any time without the need to run Setup each time, one way is to employ the concept of *virtual index* introduced in [26]. More precisely, the coefficients are modified to $\beta_{ij} = \theta_{k_2}(i \parallel id_{ij})$, where $id_{ij} = \{v_{ij}, h_{ij}\}$ is the identifier of block m_{ij}. Parameter v_{ij} is the virtual index of block m_{ij} and $h_{ij} = H_1(m_{ij} \parallel v_{ij})$, where H_1 is a secure cryptographic hash function. Virtual index of block m_{ij} is initially set to $v_{ij} = j.\delta$, where δ is a system parameter. Virtual indices determine the block orders, but as opposed to real index j, change in virtual index of one block does not affect the virtual indices of subsequent blocks. This makes the setup re-computation for the updated file more efficient at the client side.

Transparency. PoRt scheme proposed in this paper is based on the honest client assumption since it involves a keyed trusted setup. This works properly in a basic PoS scenario where a data owner wishes to outsource files to a server, and later verify the integrity of stored data. However, this assumption prevents direct application to a decentralized storage marketplace where malicious clients can collude with storage servers to collect network rewards without really storing replicas – a procedure known as generation attack. To provide security against this kind of malicious behavior, PoRt can be extended by removing the PoRt.TagGen procedure at client side by using Wesolowski's public-coin succinct argument [24]. This approach also reduces the cost of the scheme at the client side. Technically, the client executes one-time PoRt.rEncode prior to outsourcing data file. Next in the proof phase, the server executes an *aggregated* version of Wesolowski's scheme to generate a proof on correctness of time-lock puzzle (TLP) along with PoRt proof. In verification phase, verifier checks the correctness of TLP proof before verifying PoRt proof. Pseudo-codes on the right column give a detailed overview on how Extended − PoRt.Prove and Extended − PoRt.Verify algorithms work. Extended − PoRt *totally removes the client tag pre-processing* at the cost of longer proof size and increased verification cost which are only one-time costs at the end of the deposit period. Besides, to remove the trust on a single client in rEncode algorithm, multi-user encoding techniques can be employed [12]. Finally, transparency is an important research direction and we leave finding an *optimal* transparent PoRt scheme as an interesting future work.

Extended − PoRt.Prove

Input challenge $c^{(0)}$, prime ζ chosen from $Primes(\lambda)$ uniformly at random, and l encoded replicas $\{F_i\}_{1 \le i \le l}$:

Set $(\pi_1, \ldots, \pi_l) \leftarrow (1, \ldots, 1)$, $\pi_{TLP} \leftarrow 1$
Compute $q, r \in \mathbb{Z}_N$, such that $2^e = q\zeta + r$

For $k = 1$ *to* s:
  $a \leftarrow \mathcal{H}(c^{(k-1)})$
  $c^{(k)} \leftarrow a^{2^e} \in \mathbb{Z}_N$
  $\pi_{TLP} \leftarrow \pi_{TLP} \times a^q \in \mathbb{Z}_N$

  $r \leftarrow \mathcal{H}'(c^{(k)})$
  For $j \in [1, c]$
    $J \leftarrow f_r(j)$
    $y_j^{(k)} \leftarrow \mathcal{G}(c^{(k)}, j)$
  For $i = 1$ *to* l:
    $\mu_i^{(k)} \leftarrow \sum_{j \in J} y_j^{(k)} m_{ij}$
    $\pi_i \leftarrow \pi_i \times (c^{(k)})^{\mu_i^{(k)}} \mod N$
$\pi \leftarrow (\pi_1, \ldots, \pi_l)$
Return $\pi, \pi_{TLP}, \{c^{(k)}\}_{1 \le k \le s}$
Solve time-release encryption to retrieve k_1, k_2 and return them to the verifier.

Extended − PoRt.Verify

I. Verify π_{TLP} correctness:

For $k \in [1, s]$ compute $a^{(k)} \leftarrow \mathcal{H}(c^{(k-1)})$
Compute $C \leftarrow \prod_{k=1}^{s} c^{(k)}$, $A \leftarrow \prod_{k=1}^{s} a^{(k)}$
Compute $r \leftarrow 2^e \mod \zeta$
Check $C = \pi_{TLP}^{\zeta} \times A^r$

II. Verify π correctness for a file F:

Parse π as $(\pi_1, \ldots, \pi_l)$
Compute $\{\alpha_i\}_{i \in [1,l]}$, $\{\beta_{ij}\}_{i \in [1,l], j \in [1,n]}$ using k_1, k_2
Generate $tg = (tg_1, \ldots, tg_l)$ using $\{c^{(k)}\}_{1 \le k \le s}$, $\{\beta_{ij}\}_{i \in [1,l], j \in [1,n]}$.
For $i = 1$ *to* l:
  $Z_i \leftarrow (\pi_i.tg_i)^{\frac{1}{\alpha_i}}$
If $\exists i \in [1, l] : Z_i \neq majority(Z)$ return $reject$
Return $accept$