Proof of Replication
Technical Report (WIP)

Juan Benet[1] David Dalrymple Nicola Greco[1]

[1]Protocol Labs

July 27, 2017

Abstract

We introduce *Proof-of-Replication* (PoRep), a new kind of *Proof-of-Storage*, that can be used to prove that some data $\mathcal{D}$ has been *replicated* to its own uniquely dedicated physical storage. Enforcing unique physical copies enables a verifier to check that a prover is not deduplicating multiple copies of $\mathcal{D}$ into the same storage space. This construction is particularly useful in *Cloud Computing* and *Decentralized Storage Networks*, which must be transparently verifiable, resistant to Sybil attacks, and unfriendly to outsourcing.

This work (a) reviews *Proofs-of-Storage* and motivates use cases; (b) defines the novel *Proofs-of-Replication*, which can be *publicly verifiable*, *transparent*, *authenticated*, and *time-bounded*; (c) shows how to chain *Proofs-of-Replication* to establish *useful Proofs-of-Spacetime*.

Work in Progress. This is a work in progress Technical Report from Protocol Labs. Active research is under way, and new versions of this paper will appear. For comments and suggestions, contact us at research@ lecoin.io

1 Motivation and Background on *Proofs-of-Storage*

This section provides backgrounds and classifications of different *Proofs-of-Storage* and related proofs, and motivates the need for *Proofs-of-Replication*. Throughout this section, we explain the distinction between different proofs using a prover, $\mathcal{P}$, that is attempting to convince a verifier, $\mathcal{V}$, that $\mathcal{P}$ is storing some data, $\mathcal{D}$. $\mathcal{V}$ issues a challenge, c, to $\mathcal{P}$ who answers it with a corresponding proof π^c, according to the scheme in question. Proof schemes vary in their properties, their utility, and in whether $\mathcal{D}$ is useful outside the protocol or is a random string with no external utility.

1.1 Common Properties

We use the following properties, common to various proving schemes:

- (*Privately Verifiable*) A scheme is *privately verifiable* if $\mathcal{V}$ is a user with a secret verifying key generated during setup, or any other party that shares such secret key with the user. These schemes are useful in *Cloud Computing* settings, where users wish to outsource storage of data to servers and perhaps outsource verifying to a trusted verifier. As of this work, most *Proof-of-Storage* (PoS) schemes are *privately verifiable*.
- (*Publicly Verifiable*) A scheme is *publicly verifiable* if $\mathcal{V}$ can be any party with access to public data (e.g. a verifying key), but no access to the original data, or secret information generated during scheme setup. *Publicly verifiable* schemes are very useful in *Decentralized Storage Network* settings, where a verifier may be new participants who have access only to public data as context of previous proof scheme setups.

- (*Transparent*) A scheme is *transparent* if there is no extra information, sk, that enables any $\mathcal{P}$ to generate a valid proof without having data, $\mathcal{D}$. This means that there is no sk with which a malicious prover can generate proof π^* = ForgeProof(c, sk) such that 1 = Verify(c, π^*) for a $\mathcal{P}$-chosen c. *Transparent* schemes are necessary in *Decentralized Storage Networks*, where provers may also be veri ers or users. While many PoS schemes require trusting a user or veri er to generate secret keys, *Transparent* PoS schemes do not.

- (*Retrievability*) A scheme supports *retrievability* if it is possible for $\mathcal{V}$ to extract and reconstruct $\mathcal{D}$ merely by issuing many challenges c to $\mathcal{P}$ and aggregating corresponding proofs π^c. See PoRet.

- (*Dynamic*) A PoS scheme is *dynamic* if it enables the user $\mathcal{V}$ to *dynamically update* data $\mathcal{D}$ to $\mathcal{D}$' stored at server $\mathcal{P}$, to support mutable data without requiring a completely new setup. *Dynamic* PoS schemes are very useful in *Cloud Storage* settings, and systems with large, frequently mutable data without need for version history.

- (*Non-Outsourceable*) A scheme is *non-outsourceable* if $\mathcal{P}$ cannot outsource her work to some other prover $\mathcal{P}^*$ (e.g. storage, work, or proof-generation) and convince $\mathcal{V}$ that $\mathcal{P}$ did the work. *Non-outsourceable* schemes are useful in *Cloud Computing*, *Cryptocurrency*, and *Decentralized Storage Network* settings, where $\mathcal{P}$ may be rewarded for providing space, and the users or $\mathcal{V}$ wish to ensure $\mathcal{P}$ is actually providing the service.

- (*Authenticated*) A scheme is *authenticated* if the the identity of a prover can be veri ed during a proof veri cation. For example a digital signature might be required as part of generating π^c to prove identity, pk_i. *Authentication* can help make schemes *non-outsourceable*, since a prover would have to reveal secret identifying information (e.g. their private key) to the outsourced provider.

- (*Time-Bounded*) A proving scheme is *time-bounded* if a proof is only valid during a span of time. For example, some schemes declare that $\mathcal{P}$ must generate a valid proof, π, within a certain time after receiving challenge, c. If $\mathcal{P}$ delays beyond some scheme-speci c time bound, then the proof is no longer valid, as $\mathcal{P}$ had enough time to forge it.

- (*Useful*) A scheme is *useful* if it can achieve separate useful work or useful storage as part of its operation or as a side-e ect. For example, storing and verifying a veri er-chosen $\mathcal{D}$ (PDP, PoRet, PoRep) is *useful*, whereas storing and verifying a randomly-generated $\mathcal{D}$ (PoSpace) is not *useful*.

1.2 Kinds of Proofs

Here we give an overview of various kinds of proving schemes, in particular *Proofs-of-Storage* and its variants:

- ***Provable Data Possession*** (PDP) schemes [1] allow user $\mathcal{V}$ to send data $\mathcal{D}$ to server $\mathcal{P}$, and later $\mathcal{V}$ can repeatedly check whether $\mathcal{P}$ is still storing $\mathcal{D}$. PDPs are useful in cloud storage and other storage outsourcing settings. PDPs can be either *privately-verifiable* or *publicly-verifiable*, and *static* or *dynamic*. A wide variety of PDP schemes exist.

- ***Proof-of-Retrievability*** (PoRet) schemes [8], [12] are similar to PDPs, but also enable extracting $\mathcal{D}$, namely they o er *retrievability*. PDPs allow the veri er $\mathcal{V}$ to check that $\mathcal{P}$ is still storing $\mathcal{D}$, but $\mathcal{P}$ may submit valid PDP proofs yet hold $\mathcal{D}$ hostage and never release it. PoRs solves this problem by making the proofs themselves leak pieces of $\mathcal{D}$ so that $\mathcal{V}$ can issue some number of challenges and then reconstruct $\mathcal{D}$ from the proofs.

- ***Proof-of-Storage*** (PoS) schemes allow a user $\mathcal{V}$ to outsource the storage of data $\mathcal{D}$ to a server $\mathcal{P}$ and then repeatedly check if $\mathcal{P}$ is still storing $\mathcal{D}$. PDPs and PoRets were independently introduced around the same time in 2007. Since then, the concept of *Proofs-of-Storage* generalizes PDPs and PoRets. This work presents PoReps, a new type of PoS.

- ***Proof-of-Replication*** (PoRep) schemes (this work) are another kind of PoS that additionally ensure that $\mathcal{P}$ is dedicating unique physical storage to storing $\mathcal{D}$. $\mathcal{P}$ cannot pretend to store $\mathcal{D}$ twice and deduplicate the storage. This construction is useful in *Cloud Storage* and *Decentralized Storage Network*

settings, where ensuring a proper level of replication is important, and where rational servers may create Sybil identities and sell their service twice to the same user. PoRep schemes ensure each replica is stored independently. Some PoRep schemes may also be PoRet schemes.

- ***Proof-of-Work*** (PoW) schemes [6] allow prover $\mathcal{P}$ to convince verifier $\mathcal{V}$ that $\mathcal{P}$ has spent some resources. The original use case [6] presented this scheme to allow a server $\mathcal{V}$ to rate-limit usage by asking user $\mathcal{P}$ to do some expensive work per-request. Since then, PoW schemes have been adapted for use in cryptocurrencies, Byzantine Consensus [11], and many other systems. Famously, the Bitcoin network [11] expends a massive amount of energy in a hashing PoW scheme, used to establish consensus and extend the Bitcoin ledger safely.
- ***Proof-of-Space*** (PoSpace) schemes allow prover $\mathcal{P}$ to convince verifier $\mathcal{V}$ that $\mathcal{P}$ has spent some storage resources. PoSpace schemes are PoW schemes where th6 e273.206 -15(6(nsus)-421(..c(useg(nsus)-421(..c(uho0)-28(er-rec

2 *Proofs-of-Replication*

We introduce $Proof\text{-}of\text{-}Replication$ (PoRep) schemes, which allow a prover $\mathcal{P}$ to (a) commit to store n distinct $replicas$ (physically independent copies) of $\mathcal{D}$, and then (b) convince a veri er $\mathcal{V}$ that $\mathcal{P}$ is indeed storing each of the replicas. We formalize three challenges: $Sybil\ Attack$, $Outsourcing\ Attack$, and $Generation\ Attack$, all concerning $replication$. The ability to surmount these challenges distinguishes PoRep schemes from other PoS schemes. We introduce an adversarial game called RepGame that PoRep schemes must pass to be secure, and the formal de nition of PoRep schemes. Finally, we explore a sub-class of PoRep schemes, $Time\text{-}Bounded\ Proofs\text{-}of\text{-}Replication$, that are signi cantly easier to realize.

De nition 2.1. ($Sybil\ Attack$) An attacker $\mathcal{A}$ has Sybil identities $\mathcal{P}_0...\mathcal{P}_n$, and makes each commit to storing a $replica$ of $\mathcal{D}$. The attack succceeds if $\mathcal{P}_0...\mathcal{P}_n$ store less than n copies of $\mathcal{D}$ (i.e. one copy), and produce n valid $proofs\text{-}of\text{-}storage$ that convince a veri er $\mathcal{V}$ that $\mathcal{D}$ is stored as n independent copies.

De nition 2.2. ($Outsourcing\ Attack$) Upon receiving challenge c from veri er $\mathcal{V}$, an attacking prover $\mathcal{A}$ quickly fetches the corresponding $\mathcal{D}$ from another storage provider $\mathcal{P}^*$ and produces the proof, pretending that $\mathcal{A}$ has been storing $\mathcal{D}$ all along.

De nition 2.3. ($Generation\ Attack$) If attacker $\mathcal{A}$ is in a position to determine $\mathcal{D}$, then $\mathcal{A}$ may choose $\mathcal{D}$ such that $\mathcal{A}$ can re-generate $\mathcal{D}$ on demand. Upon receiving challenge c, $\mathcal{A}$ re-generates $\mathcal{D}$, and produces the proof, pretending that $\mathcal{A}$ has been storing $\mathcal{D}$ all along.

Preventing the $Generation\ Attack$ is di cult, and not usually a problem in the traditional $Cloud\ Computing$ setting, so it has not been a goal of most PoS schemes. However this attack is what prevents most PoS schemes from being used to build $Decentralized\ Storage\ Networks$, as an attacker $\mathcal{A}$ could request storage of $\mathcal{D}$ (even pay for it) and then prove storage of $\mathcal{D}$ to collect network rewards. Prior work has attempted to make it irrational to do this by adjusting fee and reward schedules but these approaches prevent useful economic constructions and do not prevent the problem. Completely preventing the $Generation\ Attack$ by forcing such attacker to store the data has been an open problem; PoRep aims solve it.

We now construct a game that tests for the three attacks together and distinguishes PoRep schemes from other PoS schemes. If a PoS scheme passes the game, then it is a PoRep scheme.

De nition 2.4. (RepGame) In the $Replication\ Game$ $\left(\{1,0\} \leftarrow \text{RepGame}(\prod^{\text{PoS}}, \quad)\right)$, an adversary, $\mathcal{A}$, with a xed amount of storage l, adaptively interacts with an honest veri er, $\mathcal{V}$, and must prove to $\mathcal{V}$ that $\mathcal{A}$ is storing n $replicas$ of data $\mathcal{D}$, such that $n = l + 1$, one more than $\mathcal{A}$ has storage space for. $\mathcal{A}$ chooses data to replicate $\mathcal{D}$, so that $\mathcal{A}$ may generate it on demand. $\mathcal{A}$ can also interact with a separate honest storage provider $\mathcal{P}$ with in nite, free storage; $\mathcal{A}$ may use $\mathcal{P}$ to only store and retrieve arbitrary data, with a latency of . $\mathcal{A}$ may also create and control any Sybil identities that $\mathcal{A}$ wishes. $\mathcal{V}$ and $\mathcal{A}$ use the $Proof\text{-}of\text{-}Storage$ proving scheme $\prod^{\text{PoS}}$. $\mathcal{V}$ asks $\mathcal{A}$ to store n di erent replicas of $\mathcal{D}$, and runs the setup PoS.Setup for each replica. $\mathcal{A}$ only has enough storage space to store l replicas of $\mathcal{D}$. Then, $\mathcal{V}$ issues a sequence of veri cation challenges c_i for each replica $i \in \{0...n\}$. $\mathcal{A}$ wins the game if she convinces $\mathcal{V}$ that she is storing all n di erent replicas, namely if $\mathcal{A}$ produces a set of valid proofs π^{c_i} that convinces $\mathcal{V}$ that PoS.Verify($\mathcal{S}_\mathcal{V}, c_i, \pi^{c_i}$) succeeds. If any call to PoS.Verify fails, $\mathcal{A}$ loses.

A PoRep is secure if there is no adversary that wins the PoRep game with more than negligible probability. Armed with these clear attacks and a game than can test for security, we can now formally de ne PoRep schemes. We use a general scheme construction, similar to the PoS construction above.

De nition 2.5. ($\prod^{\text{PoRep}}$) A general PoRep proving scheme $\prod^{\text{PoRep}}$ = (Setup, Prove, Verify) is a set of algorithms that together enable a prover $\mathcal{P}$ to convince a veri er $\mathcal{V}$ that $\mathcal{P}$ is storing a $replica$ $\mathcal{R}^\mathcal{D}$ of data $\mathcal{D}$. No two replicas $\mathcal{R}_i^\mathcal{D}$, $\mathcal{R}_j^\mathcal{D}$ can be deduplicated into the same physical storage; they must be stored independently. The three algorithms are:

- $\mathcal{R}^\mathcal{D}, \mathcal{S}_\mathcal{P}, \mathcal{S}_\mathcal{V} \leftarrow$ PoRep.Setup($1^\lambda, \mathcal{D}$), where $\mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{V}$ are scheme-speci c setup variables for $\mathcal{P}$ and $\mathcal{V}$ respectively, that depend on the data $\mathcal{D}$, and on a security parameter λ. PoRep.Setup is used to initialize

the proving scheme and give $\mathcal{P}$ and $\mathcal{V}$ information they will use to run PoRep.Prove and PoRep.Verify. Some schemes may require either party to compute PoRep.Setup, require it to be a *secure multi-party computation*, or allow any party to run it.

- $\pi^c \leftarrow$ PoRep.Prove($\mathcal{S}_\mathcal{P}, \mathcal{R}^\mathcal{D}, c$), where c is a challenge, and π^c is a proof that a prover has access to $\mathcal{R}^\mathcal{D}$ a speci c *replica* of $\mathcal{D}$. PoRep.Prove is run by $\mathcal{P}$ to produce a π^c for $\mathcal{V}$.
- $\{0,1\} \leftarrow$ PoRep.Verify($\mathcal{S}_\mathcal{V}, c, \pi^c$), which checks whether a proof is correct. PoRep.Verify is run by $\mathcal{V}$ and convinces $\mathcal{V}$ whether $\mathcal{P}$ has been storing $\mathcal{R}^\mathcal{D}$.

A PoRep must be *complete* and *secure*. In addition, PoRep schemes can be designed to have any of the properties described above in Section 1.1.

- (*complete*) if any honest prover $\mathcal{P}$ that stores a replica of $\mathcal{D}$ can always produce valid proofs that convince a veri er $\mathcal{V}$;
- (*secure*) if it can pass RepGame.

2.1 Time Bounded *Proofs-of-Replication*

There are likely to be many di erent strategies for constructing *Proof-of-Replication* protocols. Any secure construction must prevent the *Sybil Attack*, the *Outsourcing Attack*, and the *Generation Attack* and must pass the RepGame. Some protocols may be able to rely on trusted hardware while others may rely on time bounding. In this work, we are interested in creating constructions that can be deployed to existing systems and do not rely on trusted parties. We give a construction for *time-bounded* PoReps, which can be instantiated using any PDP or PoRet scheme, do not rely on trusted parties, and rely only on local time from the veri er's perspective. This construction is adaptable to *Decentralized Storage Networks*, a *publicly verifiable* setting.

Intuition for preventing the ***Sybil Attack***. Ensuring independent physical storage of n copies of $\mathcal{D}$ is similar to ensuring the storage of n di erent sets of data from which we can derive $\mathcal{D}$. We can treat each independent physical copy di erently, force a prover $\mathcal{P}$ to commit to a speci c encoding of $\mathcal{D}$ ahead of time, and check $\mathcal{P}$ is storing that speci c encoding instead. More formally, we de ne a *replica* of $\mathcal{D}$ to be an encoding using a per-replica encoding key ek: $\mathcal{R}^\mathcal{D}_{ek}$ = Encode($\mathcal{D}, ek$). Encodings must be distinguishable ($\mathcal{R}^\mathcal{D}_{ek_i} \neq \mathcal{R}^\mathcal{D}_{ek_j}$ when $ek_i \neq ek_j$) and incompressible. In order to recover $\mathcal{D}$, the encoding must be reversible: $\mathcal{D}$ = Decode($\mathcal{R}^\mathcal{D}_{ek}, ek$). Creating n di erent replicas, each with encoding key ek_i for replica $i \in 0...n$, forces any number of Sybil identities or colluding participants to prove each of those n replicas existed when producing a valid proof.

Intuition for preventing the ***Outsourcing*** and ***Generation Attacks*** We must still ensure that provers cannot get the replica just-in-time (between receiving the challenge c and producing the proof π^c), either by retrieving it from outsourced storage or by producing it by encoding $\mathcal{D}$. To achieve this, we can simply make attackers be distinguishably slower than an honest prover responding to a challenge. Computing Encode($\mathcal{D}, ek$) must take a distinguishable amount of time, such that a veri er, $\mathcal{V}$, can distinguish between:

(a) $\mathcal{T}^{\text{honest}}$ = RTT$^{\mathcal{V}\rightarrow\mathcal{P}\rightarrow\mathcal{V}}$ + Time(PoRep.Prove($\mathcal{S}_\mathcal{P}, \mathcal{R}^\mathcal{D}_{ek}, c$))
(b) $\mathcal{T}^{\text{attack}}$ = RTT$^{\mathcal{V}\rightarrow\mathcal{P}\rightarrow\mathcal{V}}$ + Time(PoRep.Prove($\mathcal{S}_\mathcal{P}$, Encode($\mathcal{D}, ek$), c))

where RTT$^{\mathcal{V}\rightarrow\mathcal{P}\rightarrow\mathcal{V}}$ is the round-trip time from $\mathcal{V}$ to $\mathcal{P}$ and back to $\mathcal{V}$. For an attacker running Encode just-in-time to be distinguishable from a slow or unlucky but honest prover, computing Encode must run distinguishably slower than acceptable variance in RTT$^{\mathcal{V}\rightarrow\mathcal{P}\rightarrow\mathcal{V}}$ and PoRep.Prove($\mathcal{S}_\mathcal{P}, \mathcal{R}^\mathcal{D}_{ek}, c$). From the perspective of $\mathcal{V}$, the wall clock running time of $\mathcal{P}$ computing Encode($\mathcal{D}, ek$) must be noticeable. Statistical estimation of RTT and PoRep.Prove to determine the acceptable variance can give a lower bound on the amount of time required. This may be required in systems where variance in RTT and PoRep.Prove do not dominate the time required for the proving step, but rather their minimums do (e.g. proofs across the interplanetary internet, or collocated in a datacenter). For distributed systems across Earth, we assume

variance in RTT to dominate, and to establish a signi cant bound we set the minimum time of Encode $\mathcal{T}^{\text{Encode}} = 10 \times \mathcal{T}^{\text{honest}}$ or $100 \times \mathcal{T}^{\text{honest}}$. As long as veri ers can clearly distinguish $\mathcal{T}^{\text{honest}} << \mathcal{T}^{\text{attack}}$ we defeat both the $Outsourcing$ and $Generation\ Attacks$.

Intuitions for Encode. The remaining question is what function should Encode be? We have explored some of the requirements: (a) it must be slow enough to distinguish $\mathcal{T}^{\text{honest}} << \mathcal{T}^{\text{attack}}$; (b) it must be reversible; (c) the output should be determined from an encoding key; and (d) information should not be compressible across replicas. Additional design goals follow: (a) Decode should allow fast recovery of $\mathcal{D}$, ideally the running time of Decode grows sub-linearly in respect to Encode; (b) Encode's running time should scale arbitrarily with a tunable parameter; (c) Encode should be publicly veri able, so that anybody, including $\mathcal{P}$, can verify it; (d) Encode should use all of $\mathcal{D}$ multiple times before nishing, so that a partial replica cannot be computed from partial $\mathcal{D}$, which would a ect timing assumptions; (e) Encode should support high-entropy encoding keys; and (f) a slight variation (single bit ip) in the encoding key should change the encoding completely. What we need is a PRP ($pseudo\text{-}random\ permutation$) that matches our desires.

Slowable PRPs. It is easy to construct a PRP that can be slowed down arbitrarily by using a $block\ cipher$ (BC) in $cipher\ block\ chaining$ (CBC) mode. The following three variations of CBC sequentially increase in complexity to achieve the desired properties:

- ($streaming$) We simply run a modi ed version of CBC.Encrypt that encrypts each block τ times before moving on to the next. This slows down the encryption by a factor of τ, making it run in $\mathcal{O}(n\tau)$ time, where n is the size of the input data. Note that encryption is still sequential; this is important to slow down encryption cheaply, in that decryption is still parallelizable ($\mathcal{O}(n\tau)$ time, with up to τ parallelization), which speeds up recovery of the original data and allows random reads. The only issue with this scheme is that it is $depth\text{-}first$: it computes and outputs each cipher text block before reading the next input block (i.e. streaming process). This makes it possible to encrypt having only a partial input le, and to start using the encrypted output after each iteration, signi cantly lowering the slow-down.
- ($shuffling$). To x the issues with the $streaming$ scheme above we can try some random shu ing: before and after starting CBC.Encrypt, apply a random shu e that is likely to disperse information across all the blocks, making it impossible to use sequential segments on an encrypted block until the whole data is encrypted. This also a ects decryption, as a decryptor must have the entire le and de-scramble it before running the streaming and parallelizable CBC.Decrypt.
- ($layering$) Another version runs all of CBC.Encrypt for τ iterations, taking the output of the last iteration as the input of the next, making sure to chain across iterations. This chaining preserves the sequential property of CBC encryption: the last cipher block of iteration $\tau - 1$ is chained with the rst block of iteration τ. This sequentiality ensures the encryption cannot be parallelized. The running time of CBC.Encrypt is still $\mathcal{O}(n\tau)$ and CBC.Decrypt is still parallelizable ($\mathcal{O}(n\tau)$ with up to τ parallelization), but requires decrypting in whole layers before recovering any plaintext.

We can make any of these constructions $publicly\ verifiable$ without the data (cheaply) by computing them within a SCIP scheme (SNARKs, STARKs) [7, 4, 3, 2]. Doing this would be expensive, and it is likely primitives used in SCIP scheme constructions can be used directly in a cheaper way. We believe this is an open problem.

De nition 2.6. (Seal) We de ne a pair of encoding functions Seal and Unseal, inverses of each other, and parametrized over any secure $block\ cipher$ algorithm BC and number of rounds t:

$$\text{Seal}^{\tau}_{\text{BC}}(ek, \mathcal{D}) := \text{BC.CBC.Encrypt}^{\tau}(ek, \mathcal{D})$$
$$\text{Unseal}^{\tau}_{\text{BC}}(ek, \mathcal{R}^{\mathcal{D}}_{ek}) := \text{BC.CBC.Decrypt}^{\tau}(ek, \mathcal{R}^{\mathcal{D}}_{ek})$$

where BC.CBC.$\{$Encrypt, Decrypt$\}$ are the Encrypt,Decrypt functions of BC in CBC mode; τ is the number of encryption rounds to perform, which is chosen to slow down Seal; and $\{\text{Encrypt}^{\tau}, \text{Decrypt}^{\tau}\}$ is the recursive application of Encrypt or Decrypt for τ iterations, where the last cipher block of each iteration is chained back onto the rst block of the following iteration, ensuring that the entire encryption is sequential.

With $\text{Seal}^{\tau}_{\text{BC}}$ and any existing PDP or PoRet scheme, we can construct a *time bounded* PoRep scheme, where the proofs prove that the storage of a speci c *replica* is encoded with $\text{Seal}^{\tau}_{\text{BC}}$. The scheme is *time bounded* because the proofs only prove possession of the *replica* at the time of the challenge if they are produced by $\mathcal{P}$ and checked by $\mathcal{V}$ in less time than the running time of $\text{Seal}^{\tau}_{\text{BC}}$.

De nition 2.7. ($\prod^{\text{SealPoRep}}$) A *time-bounded* Seal-based PoRep proving scheme $\prod^{\text{SealPoRep}}$ = (Setup, Prove, Verify) is a set of algorithms that together enable a prover $\mathcal{P}$ to convince a veri er $\mathcal{V}$ that $\mathcal{P}$ is storing independent *replica* $\mathcal{R}^{\mathcal{D}}_{ek}$ of data $\mathcal{D}$, where $\mathcal{R}^{\mathcal{D}}_{ek}$ = $\text{Seal}^{\tau}_{\text{BC}}(ek, \mathcal{D})$ for some encoding key ek, a secure block cipher BC, and a timing parameter τ. These algorithms rely on another *Proof-of-Storage* proving scheme $\prod^{\text{PoS}}$, which is used internally to prove storage of the replica, and may be either a PDP or PoRet scheme. The three algorithms are:

- $\mathcal{R}^{\mathcal{D}}_{ek}, ek \leftarrow$ SealPoRep.Setup($1^{\lambda}, \tau, \mathcal{D}$) where ek is an encoding key unique to a single setup. ek may be chosen randomly, or derived from the identity of $\mathcal{P}$ and a unique replica number. SealPoRep.Setup is used to initialize the proving scheme: to choose an encoding key ek, to compute $\mathcal{R}^{\mathcal{D}}_{ek}$ = $\text{Seal}^{\tau}_{\text{BC}}(ek, \mathcal{D})$, and to compute PoS.Setup($1^{\lambda}, \mathcal{R}^{\mathcal{D}}_{ek}$), all of which $\mathcal{P}$ and $\mathcal{V}$ will use to run SealPoRep.Prove and SealPoRep.Verify. With a *publicly verifiable* $\text{Seal}^{\tau}_{\text{BC}}$, any party can run the setup, even $\mathcal{P}$. The τ timing parameter must be chosen such that running $\text{Seal}^{\tau}_{\text{BC}}$ is distinguishibly more expensive than just proving and verifying.
- $\pi^c \leftarrow$ SealPoRep.Prove($\mathcal{R}^{\mathcal{D}}_{ek}, c$) where c is a challenge, and π^c is a proof that a prover has access to $\mathcal{R}^{\mathcal{D}}_{ek}$. SealPoRep.Prove computes π^c = PoS.Prove($\mathcal{R}^{\mathcal{D}}_{ek}, c$). It is run by $\mathcal{P}$.
- $\{0,1\} \leftarrow$ SealPoRep.Verify(c, π^c) which checks whether a proof is correct. SealPoRep.Verify internally computes $\{0,1\} \leftarrow$ PoS.Verify(c, π^c). It is run by $\mathcal{V}$, and convinces $\mathcal{V}$ whether $\mathcal{P}$ has been storing $\mathcal{R}^{\mathcal{D}}_{ek}$

3 *Proofs-of-Spacetime*

Usually, most PoS and PoSpace schemes are described in terms of interactive challenge settings, where a veri er $\mathcal{V}$ issues single or periodic challenges to a prover $\mathcal{P}$. Unpredictable yet frequent challenges can give $\mathcal{V}$ con dence that $\mathcal{P}$ has been correctly storing the required data $\mathcal{D}$ for the duration of time challenged. These challenges must be frequent and unpredictable, as too infrequent or predictable challenges could give $\mathcal{P}$ a chance to cheat by retrieving $\mathcal{D}$ (from other outsourced storage or from a secondary market) in advance of $\mathcal{V}$'s challenge. $\mathcal{V}$'s con dence increases with the frequency of challenges. This is a useful way for a *Cloud Storage* client ($\mathcal{V}$) to ensure a particular service ($\mathcal{P}$) is correctly storing the client's data over time. However, these interactive checks require the clients to be online and spend bandwidth and computation resources during all challenges.

Time-bounded PoReps could allow the time intervals to be large, as the Seal function can be scaled to be signi cantly larger than the maximum desired time between intervals. Yet, this would not obviate the need for periodic challenges; many challenges would still be needed over long periods of time, and these would require $\mathcal{V}$ to be online and spending resources to perform them. In both *Cloud Storage* and *Decentralized Storage Network* settings, it would be quite useful to allow clients to check proofs infrequently, perhaps coupled with other necessary accesses, such as retrieving $\mathcal{D}$. It would also be useful to have an auditable record that shows $\mathcal{P}$ has been behaving correctly { storing $\mathcal{D}$ { for the entire duration of time. This record could even be time-stamped into public ledgers such as blockchains.

Chaining Sequential Proofs. We can construct a sequence of challenges and proofs where the challenge at iteration n (c_n) is derived deterministically from the proof at iteration $n-1$ (π_{n-1}). Let a *proof-chain* be these sequential challenges and proofs stored together, for a veri er to inspect all at once. Note that every proof must be correctly produced; if any proof fails to verify, the whole chain fails to verify.

De nition 3.1. (*proof-chain*) A *proof-chain* is a veri able data-structure that chains together a sequence of challenges and proofs. For iteration n, let c_n be a challenge, π_n be a proof for c_n, and $\mathcal{C}_n$ be the *proof-chain* formed by extending $\mathcal{C}_{n-1}$ with π_n. Given a proof scheme with functions (Prove, Verify), randomness r, and a *collision resistant hash-function* (CRH) $\mathcal{H}$; we can construct the data structure and verify it with

VerifyChain, according to the following rules:

- $c_0 \leftarrow \mathcal{H}(r)$
- $c_n \leftarrow \mathcal{H}(n||\pi_{n-1})$
- $\pi_n \leftarrow$ Prove(c_n)
- $\mathcal{C}_0 \leftarrow (c_0, \pi_0)$
- $\mathcal{C}_n \leftarrow (\mathcal{C}_{n-1}, \pi_n)$
- $\{0,1\} \leftarrow$ VerifyChain($\mathcal{C}_0$) = Verify(c_0, π_0)
- $\{0,1\} \leftarrow$ VerifyChain($\mathcal{C}_n$) = VerifyChain($\mathcal{C}_{n-1}$) & Verify(c_n, π_n)

Time and ***proof-chains***. Such a $proof\text{-}chain$ would be a verifiable record of a prover producing proofs over a duration of time. There are schemes with predictable computation time, where a verifier $\mathcal{V}$ can know how long it takes a prover $\mathcal{P}$ to produce each iteration (i.e. with some well-known and bounded variance). Using such schemes, $\mathcal{V}$ could ask $\mathcal{P}$ to begin producing a chain at time t_0. At a future time t_n, $\mathcal{V}$ can ask $\mathcal{P}$ to return the chain, which should contain as many entries as can be computed within the interval $t_n \rightarrow t_0$. If the chain passes verification, $\mathcal{V}$ knows that $\mathcal{P}$ has spent that interval of time ($t_0 \rightarrow t_n$) producing the proof chain. This also implies $\mathcal{P}$ has been storing $\mathcal{D}$, or at least has had sufficiently fast access to $\mathcal{D}$, during the entire interval. $\mathcal{V}$ can have the same degree of confidence as with a sequence of online PoS challenges. Since producing the $proof\text{-}chain$ is a sequential process, with parallelization only possible in the proof algorithm itself, $\mathcal{P}$ need only provide enough processing power to compute a single proof at a time, and $\mathcal{P}$ cannot spend more computation resources in parallel to speed up the process and cheat the expectations. The nth iteration of the $proof\text{-}chain$ has similar guarantees as an equivalent online PoS challenge.

Reducing disk accesses and time variance. The process of computing such a $proof\text{-}chain$ would force a prover $\mathcal{P}$ to be constantly running a PoS proving process, which may require significant computation resources and disk accesses. This should not be a task too onerous for most modern computers to perform. However, some use-cases may need to reduce the amount of disk accesses, or to reduce the variance in the time required to compute each iteration of the $proof\text{-}chain$. To do so, we can adjust the chaining scheme to run some additional $sequential$ computation when deriving the next challenge. This computation can be tuned to have the desired properties: low time variance, no disk accesses, CPU bounded, memory bounded, storage bounded, etc. For example, the Sloth hash function [9] is arbitrarily scalable in time, is CPU bounded, has low time variance, and makes zero disk accesses. The Balloon hash function [5] is arbitrarily scalable in time, is memory bounded, has low time variance, and makes zero disk accesses. These are only two examples of sequential processes with predictable running times can be used to tune the properties of the $proof\text{-}chain$ computation.

Proof-chains and spacetime. The creation of this sequential $proof\text{-}chain$ of PoS or PoSpace proofs implies that a specific amount of space was kept occupied storing data $\mathcal{D}$ for a specific duration of time. In other words, a slice of $spacetime$ was committed to storing the relevant data $\mathcal{D}$. Hence we call this chaining of sequential proofs a $Proof\text{-}of\text{-}Spacetime$ (PoSt). These proofs are inherently $time\text{-}bounded$, in that a PoSt proof for a duration of time t is only valid if verified shortly after, but the chain can be extended arbitrarily. The only bound is the variance in the time to compute each iteration. The variance of computing the whole PoSt $proof\text{-}chains$ must be low enough to have confidence over the time interval.

PoSt to the blockchain. The sequential PoSt $proof\text{-}chains$ can get arbitrarily long when involving time-stamping services { such as the Bitcoin blockchain. Given a time-stamping service trusted by both $\mathcal{P}$ and $\mathcal{V}$, $\mathcal{P}$ can periodically time-stamp the head of the $proof\text{-}chain$ (creating a checkpoint), effectively anchoring the $proof\text{-}chain$ in time. This can happen while $\mathcal{V}$ is offline. In the future, $\mathcal{V}$ can decide to audit the $proof\text{-}chain$ and its time-stamped checkpoints, and verify that $\mathcal{P}$ correctly produced the $proof\text{-}chain$ during the right intervals of time.

De nition 3.2. ($\prod^{\mathsf{PoSt}}$) A general PoSt proving scheme $\prod^{\mathsf{PoSt}}$ = (Setup, Prove, Verify) is a set of algorithms that together enable a prover $\mathcal{P}$ to produce an incremental *time-bounded proof-chain* $\mathcal{C}_n$ which proves to a veri er $\mathcal{V}$ that $\mathcal{P}$ has been storing data $\mathcal{D}$ during iterations $0 \rightarrow n$ of the *proof-chain*. The *proof-chain* must be periodically checked or time-stamped. These algorithms rely on a *Proof-of-Space* (PoSpace) proving scheme $\prod^{\mathsf{PoSpace}}$, which may also be a *Proof-of-Storage* scheme ($\prod^{\mathsf{PoS}}$). The three algorithms are:

- $\mathcal{D}, c_0, \mathcal{S}_\mathcal{P}, \mathcal{S}_\mathcal{V} \leftarrow$ PoSt.Setup($1^\lambda, \mathcal{D}$) where c_0 is an initial random challenge not controlled by $\mathcal{P}$, $\mathcal{S}_\mathcal{P}$ and $\mathcal{S}_\mathcal{V}$ are scheme-speci c setup variables for $\mathcal{P}$ and $\mathcal{V}$ respectively, that depend on the data $\mathcal{D}$, and on a security parameter λ. PoSt.Setup is used to initialize the proving scheme and give $\mathcal{P}$ and $\mathcal{V}$ information they will use to run PoSt.Prove and PoSt.Verify. Some schemes may require either party to compute PoSt.Setup, require it to be a *secure multi-party computation*, or allow any party to run it.
- $\mathcal{C}_{n+1} \leftarrow$ PoSt.Prove($\mathcal{S}_\mathcal{P}, \mathcal{D}, \mathcal{C}_n$) where $\mathcal{C}_n$ and $\mathcal{C}_{n+1}$ are *proof-chains* that prove $\mathcal{P}$ had access to $\mathcal{D}$ up to iterations n and $n+1$ respectively. PoSt.Prove is run by $\mathcal{P}$ to produce each *proof-chain* iteration, extending the chain with the next proof (π_{n+1}), as computed by PoSpace.Prove. Note that the challenges and proofs are computed according to the proving scheme and the *proof-chain* construction. The *proof-chain* and its proofs will later be checked by $\mathcal{V}$.
- $\{0,1\} \leftarrow$ PoSt.Verify($\mathcal{S}_\mathcal{V}, \mathcal{C}_n$) which checks whether *proof-chain* is correct, by verifying every iteration's proof with PoSpace.Verify. PoSt.Verify is run by $\mathcal{V}$ and convinces $\mathcal{V}$ whether $\mathcal{P}$ has been storing $\mathcal{D}$ during iterations $0 \rightarrow n$.

4 Realizable Constructions (TODO)

4.1 Realizable time-bounded *Proof-of-Replication*

SealPoRep with $\mathsf{Seal}^{\tau}_{\mathsf{AES}-256}$, and τ such that $\mathsf{Seal}^{\tau}_{\mathsf{AES}-256}$ takes 10-100x the honest prover/veri er time.

4.2 Realizable useful *Proofs-of-Spacetime*

A PoSt with the PoRep from 5.1, checked or time-stamped frequently (into a blockchain).

Acknowledgements

This work is the cumulative e ort of multiple individuals within the Protocol Labs team, and would not have been possible without their the help, comments, and review.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 598{609, New York, NY, USA, 2007. ACM.

[2] E. Ben-Sasson, I. Bentov, A. Chiesa, A. Gabizon, D. Genkin, M. Hamilis, E. Pergament, M. Riabzev, M. Silberstein, E. Tromer, et al. Computational integrity with a public random string from quasi-linear pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 551{579. Springer, 2017.

[3] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90{108. Springer, 2013.

[4] N. Bitansky, A. Chiesa, and Y. Ishai. Succinct non-interactive arguments via linear interactive proofs. Springer, 2013.

[5] D. Boneh, H. Corrigan-Gibbs, and S. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 220{248. Springer, 2016.

[6] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139{147, London, UK, UK, 1993. Springer-Verlag.

[7] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626{645. Springer, 2013.

[8] A. Juels and B. S. Kaliski, Jr. Pors: Proofs of retrievability for large les. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584{597, New York, NY, USA, 2007. ACM.

[9] A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx.

[10] T. Moran and I. Orlov. Proofs of space-time and rational proofs of storage. Cryptology ePrint Archive, Report 2016/035, 2016. http://eprint.iacr.org/2016/035.

[11] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[12] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '08, pages 90{107, Berlin, Heidelberg, 2008. Springer-Verlag.