PoReps: Proofs of Space on Useful Data

Ben Fisch

Stanford University, Protocol Labs

Abstract

A *proof-of-replication* (PoRep) is an interactive proof system in which a prover defends a publicly verifiable claim that it is dedicating unique resources to storing one or more retrievable replicas of a data file. In this sense a PoRep is both a *proof of space* (PoS) and a *proof of retrievability* (PoR). This paper establishes a foundation for PoReps, exploring both their capabilities and their limitations. While PoReps may unconditionally demonstrate possession of data, they fundamentally cannot guarantee that the data is stored redundantly. Furthermore, as PoReps are proofs of space, they *must* rely either on rational time/space tradeoffs or timing bounds on the online prover's runtime. We introduce a rational security notion for PoReps called ϵ-rational replication based on the notion of an ϵ-Nash equilibrium, which captures the property that a server does not gain any significant advantage by storing its data in any other (non-redundant) format. We apply our definitions to formally analyze two recently proposed PoRep constructions based on verifiable delay functions and depth robust graphs. Lastly, we reflect on a notable application of PoReps—its unique suitability as a Nakamoto consensus mechanism that replaces proof-of-work with PoReps on real data, simultaneously incentivizing and subsidizing the cost of file storage.

1 Introduction

A *proof-of-replication* (PoRep) builds on the two prior concepts of proofs-of-retrievability (PoR) [30] and proofs-of-space (PoS) [24]. In the former a *prover* demonstrates that it can retrieve a file and in the latter the prover demonstrates that it is using some minimum amount of space to store information. Most proofs of space require the prover to use this space to store junk information that is only relevant to the PoS protocol. A PoRep, in essence, embeds a PoR within a PoS. It enables the prover to demonstrate that it is using some minimum amount of space while simultaneously allowing it to actually use that space to store useful information. An additional critical property of a PoRep is that the storage costs required to succeed in the protocol depend only on the *size of the data inputs* and are otherwise independent of the data inputs. In particular, the cost to succeed in the protocol should not depend on whether or not this data was privately preprocessed (e.g. encrypted by a client) or generated by the server itself. As another special case, if the input to the protocol were k redundant copies of the same file then this would cost the same as running the protocol on k distinct data files. Intuitively, this would achieve the following property: even if a PoRep prover *could* pass the protocol without storing the data redundantly (e.g. by deduplicating the k copies), there would be no advantage to doing so. In other words, it would be *rational* for a PoRep prover in this scenario to honestly store k copies of the data.

Consider as a thought exercise a simple composition of a PoR protocol and PoS protocol that would not succeed in achieving these goals. This simple protocol requires the prover to use a total of $2N$ space. The prover uses half of this space to produce a PoS (i.e. it runs a standard PoS protocol that requires it to fill this space with random data) and it uses the other half to actually store some useful data file of size N and produce PoRs of the file. This satisfies both a PoR and a PoS with only $2N$ storage and therefore shows both that the prover is using some minimum $\Omega(N)$ amount of storage and is able to retrieve the data of interest. However, it fails the "independent cost" criterion. Namely, it is more expensive for the prover to run this protocol on useful data (requiring $2N$ space to store both the useful data and the random data) than to just store the useless random data required for the PoS and provide a PoR for this random data (requiring only N space). Moreover, consider if the prover were asked to store k redundant copies of the same file D and use this protocol to prove that it (a) is using at least kN space to store these copies and (b) is able to retrieve D. Following the protocol honestly requires $2kN$ space: kN space to store the k copies of D and kN space to store the random data required for the PoS. Unfortunately, the most rational strategy would be to store only the random data and a single copy of D as this uses only $(k+1)N$ space and still allows the prover to successfully pass the protocol.

While standard PoRs can provide proofs of data duplication in a private-verifier[1] setting where the client preprocesses its own data before sending it to the server, their security relies on a non-colluding client to privately preprocess the data. One advantage of PoReps over standard PoRs for proofs of data duplication is that multiple clients could contribute data to a single database and would not need to trust any single client to preprocess the data. PoReps could also be used to provide proofs of storage for publicly available data. For example, a consensus server in a massively distributed and open state-replication system such as Bitcoin could provide a PoRep that it is storing a complete history of the state-machine transcript (i.e. in *blockchain* systems like Bitcoin this is referred to as a "full node" storing the "chain"). Unlike PoRs, PoReps could be used to provide this proof without requiring all the verifiers to send the server their own preprocessed copy of the public transcript (incurring impractical communication).

PoReps provide a publicly verifiable proof of data duplication, secure against adversaries who will not deviate from an ϵ-rational honest strategy. The notion of an ϵ-equilibrium is used in game theory as a generalization of a Nash equilibrium where players gain at most an ϵ advantage from deviating. This solution concept is appropriate for a malicious-but-lazy adversary, or in conjunction with the status-quo-bias assumption: you are on the couch and the TV remote is across the room, so you continue to watch the same channel. This may seem like a strangely weak security property to achieve in a cryptographic protocol. For a very simple reason it is actually the best possible security that PoReps can achieve, at least in the standard[2] model of computation for cryptographic analysis. In short, any prover storing k independent replicas of the file could intentionally *correlate* these replicas in a way that it can still efficiently retrieve each in its original format. For example, it could encrypt them and store the key.

However, the primary use case of PoReps is probably not a proof of storage system that weakly discourages de-duplication. The properties achieved by PoReps make it uniquely suited

---

[1]Verification of PoRs can also be outsourced. Proofs of retrievability where a client can outsource the work of verifying proofs were previously referred to as publicly verifiable PoRs or "public PoRs", however the label *outsourceable* appropriately distinguishes their security property from the public verifiability of PoReps and proofs of space.

[2]This does not preclude secure hardware solutions or proofs based on network timing.

for a Nakamoto consensus mechanism (also known as *blockchain* consensus) that use PoReps as a *useful proof of space* in place of Bitcoin's proof-of-work to achieve sybil-resistance. Briefly, Nakamoto consensus mechanisms and its variants are a special type of state-machine replication process managed by an unpermissioned, asynchronous, and distributed network of consensus participants, with the additional feature that the state-machine itself encodes fungible tokens of value. In particular, a defining characteristic of these consensus mechanisms is their ability to mint new tokens in the state machine in order to reward and incentivize consensus participants (termed *miners*), taking for granted that these tokens represent real-world assets.

Blockchain systems based on proof-of-space have been proposed [19, 42] and are in active development, pursuing goals including energy efficiency and more egalitarian distribution.[3] PoReps target a different advantage entirely: they simultaneously incentivize useful peer-to-peer data storage. This is the basis of Filecoin [1]. The miners that manage the system's distributed state-machine are required to produce PoReps in order to append transactions, or equivalently, be elected as temporary consensus leaders, and are rewarded with freshly minted coins in return. Accepting the hypothesis that the miners will be incentivized by these rewards alone to produce PoReps, just as Bitcoin miners produce wasteful proofs-of-work, the rewards–in effect–subsidize the auxiliary *useful work* accomplished by the PoRep: file storage.

The utility of ϵ-rational replication in this context is immediately transparent. It characterizes the ϵ cost required to nudge a data replication strategy from a weak equilibrium strategy into a strong one. In other words, it represents the cost that clients must pay (in a stylized model that ignores other market variables) to convince miners to encode their real data inside PoReps rather than "useless" generated data, and therefore the degree to which a system such as Filecoin subsidizes storage costs. PoReps thus sit at an exciting crossroads of economics and cryptography:

> *PoReps epitomize a cryptographic mechanism that is concerned not solely with the actions and capabilities of an adversary, nor penalties levied for misbehavior, but rather its broader effect on behaviors in an economy.*

1.1 Related work

Proofs of storage   Cryptographic proofs of storage have been proposed in a variety of flavors throughout the literature. Broadly speaking, proofs of storage are interactive protocols between a server (prover) and client (verifier) with different goals relating to statements about the server's storage. Proofs-of-retrievability (PoR) [30] demonstrate that the prover can retrieve some specified data known to the verifier. A *public-key* PoR is outsourceable, meaning that the client can provide a public authentication tag that any verifier can use to verify the server's proofs. Unless the data itself is incompressible, a PoR does not necessarily prove anything to the verifier about how much space the prover is using in order to retrieve the file. In particular, in the public-key setting the third party verifier does not know whether the client is colluding with the server or not, and therefore a PoR is not a truly publicly verifiable claim of storage. In proofs-of-space (PoS) [24] a time-constrained prover demonstrates that it is storing some incompressible string of $\Omega(N)$ bits, and therefore using at least $\Omega(N)$ bits of space. This formulation of PoS is sometimes referred to as a *proof of persistent space* as opposed to a proof of transient space [46]. Timing assumptions are inherently required in proofs of persistent space due to the efficiency

---

[3]The Chia Network is a blockchain system based on PoS, `https://chia.net/`.

requirement that the communication between the prover and verifier is required to be compact, as otherwise the prover can store only the communication transcript and recompute the advice. Proofs-of-secure-erasure (PoSE) [44] are protocols similar to a PoS intended to demonstrate that a storage-bound prover has erased all of its storage. An important difference between PoSE and the standard PoS notion is that they involve a private-key verifier who preprocesses data, similar to a PoR proof. They were suggested as a protocol for a memory-bound embedded device to prove to a server that it has erased private data or all prior code before an update, e.g. to ensure that any malware has been erased.

Rationally secure storage protocols   All of the above proof of storage protocols are unconditionally secure, or at least characterize security in terms of strong time-space lower bounds. However, there are additionally a class of proof of storage protocols that are secure in a weaker *rational adversary* model. The model considered in all of these works is an adversary that acts to minimize its storage costs. Storage enforcing commitments [28] were a precursor to PoR that require the prover to dedicate a minimum amount of storage in order to pass the protocol for a committed file F (preprocessed by a client with a secret key). Their constructions did not guarantee unconditionally that the prover was actually storing F, and were therefore only secure in an epsilon-rational sense. Hourglass schemes [51] were proposed as a generic method to prove that a server is storing data encoded in a specified format, in particular for the use cases of encryption-at-rest and file watermarking. Similarly, they leave open the possibility that the server is also storing a plaintext or non-watermarked version of the file. It only guarantees that such a server must use twice the storage resources to act maliciously.

Proof of data replication vs PoReps   A proof of data replication/duplication, specifically in the sense of proving that data is stored in a redundant format, is a special case of hourglass schemes. Proofs of data replication/duplication have been explored primarily in the private-client setting similar to proofs of retrievability [7,47]. In this setting a client is somehow involved in preprocessing the data and either the client is the verifier or the client outsources its own verification work. The system Mirror [7] is one such example, which proposed a way for the server to replicate the client's initially preprocessed file without requiring further communication from the client. The main technique in Mirror is a variant on RSA time-lock puzzles, and the security relies on timing assumptions similar to PoS. Several days ago another paper exploring this line of work was posted, which proposed a construction of proof of data replication in the private-client setting without timing assumptions [21].

   The term proof-of-replication (PoRep), which is the focus of our work, was proposed by the Filecoin project [2] in reference to a hybrid of a PoR and PoS that demonstrates in a publicly verifiable way that the prover is using space to store a unique replica of a file, or even several unique replicas of the same file. The main intent of this proposed primitive was to resist the so called "generation attack", i.e. it should be a publicly verifiable proof of space even when the server may collude with a client. A nearly equivalent concept to proof-of-replication was proposed earlier by Sergio Demian Lerner in 2014 under the name *proof of unique blockchain storage* [33]. It focussed on the special case where all provers/verifiers in the system know at least one copy of the replicated file. For instance, it was proposed as a mechanism for demonstrating publicly that the Bitcoin blockchain is highly replicated. Lerner proposed using time-asymmetric encodings to apply a slow transformation to a file using a unique identifier as a key. Boneh et. al. generalized this construction using a new primitive called a decodable verifiable delay function

(VDF) [14]. New constructions of PoReps using more traditional tools from proofs of space (i.e. hard-to-pebble graphs) were concurrently proposed in [26, 45].

Tight proofs of space As we develop in this work, there is a strong connection between tight proofs of space and PoReps. A tight proof of space is one where any adversarial prover implementing an attack uses only an ϵ fraction less space than the honest prover where ϵ can be made arbitrarily small. Intuitively, if a PoRep is not a tight proof of space (e.g. $\epsilon = 1/2$) then there may be some adversary that would be rationally incentivized to deviate from honest behavior and therefore likely destroy the replication format. Pietrzak [45] constructed the first tight PoS, although the proof size is $O(1/\epsilon^2)$ and depended on depth robust graphs with rather special properties [6]. Concretely, according to the analysis it requires graphs of degree $2,760 \log N$ just to achieve $1/2$ space gap. The PoRep based on verifiable delay encodings as a moderately hard function [14] is in fact an arbitrarily tight PoS, however it is exceedingly inefficient to generate on large data (scaling quadratically in the data size).

1.2   Contributions and Organization

This work is a foundational study of PoReps. Primarily, we motivate the goals of PoReps and explore what they are in contrast to other cryptographic primitives. Although many of the related primitives including proofs of retrievability, proofs of space, and proofs of data replication (private-client setting) were formally defined and studied in prior works, this is the first comprehensive formal study of PoReps. While a presentation at BPASE 2018 [26] provided the first basic definitions for PoReps and intuition about both the impossibility of ideal security and how to define ϵ-rational replication, we build upon this further and provide a comprehensive model and definitions. In particular, our framework also addresses issues of composable security noted in a recent paper by Fisch and Silas [27]—namely the issue that the prover may be incentivized to deviate from a rational replication strategy if deviating somehow benefits another simultaneous storage mechanism (e.g., perhaps it could use the PoRep data to compress auxiliary data that it is storing).

Relevant background information and preliminary tools are discussed in Section 2. While for the most part this is a review, we also present a new formulation of a primitive we call *proof of retrievable commitment* (PoRC) and a simple generic construction based on vector commitments. This generalizes the simple public proof of retrievability on Merkle commitments that is used ubiquitously, particularly as a subprotocol in proofs of space. Section 3 defines PoReps abstractly and presents our security model for ϵ-rational replication (Definition 23). We show that this definition implies that the PoRep is a PoS, although it does not demand it explicitly. Proving that a PoRep construction satisfies the definition also likely requires "knowledge of compression assumptions" [27]. We formulate one such assumption along with a key lemma (Lemma 4) that establishes sufficient criteria for ϵ-rational replication, showing that if a PoRep construction satisfies PoRC and PoS in a certain sense then it will also satisfy ϵ-rational replication. This greatly simplifies the analysis of candidate PoRep constructions. Section 4 presents in detail the two constructions Basic-VDE-PoRep from VDFs [14] and DRG-PoRep [26] that fuses more traditional techniques in PoS (hard-to-pebble DAGs) with VDFs. In Section 5 we discuss practical instantiations of DRGs and some new experimental measurements regarding their security. Finally, Section 6 applies the model, definitions, and Lemma 4 of Section 3 to formally analyze and prove secure both constructions DRG-PoRep and Basic-VDE-PoRep.

Contents

2   Preliminaries and Background

2.1   Compression and Erasure Codes

A *source* $\mathcal{S}$ over $\{0,1\}^n$ is a distribution over strings in $\{0,1\}^n$. Encoding schemes consist of a pair of algorithms (enc, dec) such that enc maps strings from the support $\mathrm{Sup}(\mathcal{S})$ of a source $\mathcal{S}$ to a code space Σ^m over an alphabet Σ, and dec recovers inputs in $\mathrm{Sup}(\mathcal{S})$ from codes in Σ^m. If $\mathrm{Sup}(\mathcal{S}) = \{0,1\}^n$ and the distribution $\mathcal{S}$ is not relevant to the properties of (enc, dec) then we

may just call it an encoding scheme on $\{0,1\}^n$. The two types of encoding schemes of interest here are compression and erasure codes, which have opposite goals.

Erasure codes Erasure codes, as the name suggests, can tolerate deletion of a constant fraction of symbols in a code and still recover the encoded data. They are an extreme form of redundant storage.

Definition 1. *An (r, δ) erasure code is an encoding scheme (enc, dec) on $\{0,1\}^n$ to Σ^n where $n/m = r$ (called the rate) and for all $x \in \{0,1\}^n$ such that $\mathsf{enc}(x) = (w_1, ..., w_m)$ the decoding algorithm dec can recover x from any $(1-\delta)m$ symbols in w, i.e. if $E \subset [m]$ is any ϵm size subset and $\hat{w} = (\hat{w}_1, ..., \hat{w}_m)$ where $\hat{w}_i = \perp$ for $i \in E$ and $\hat{w}_i = w_i$ for all $i \notin E$ then $\mathsf{dec}(\hat{w}) = x$. A $(1-\delta, \delta)$ erasure code is called an optimal rate $1 - \delta$ erasure code.*

Compression schemes The goal of compression, in contrast, is to encode the original data source with the smallest possible number of symbols such that the data can still be decoded efficiently (at least with high probability). Often a compression scheme will output codes of variable length depending on the input, however for simplicity the following definitions and facts focus on the special case that all outputs are padded to the same length. Unlike erasure codes, compression schemes are not concerned with structural properties of the code other than its overall length, hence without loss of generality the alphabet is $\Sigma = \{0,1\}$.

Definition 2 (Compression scheme)**.** *An $(\mathcal{S}, k)$-compression scheme for a source $\mathcal{S}$ over $\{0,1\}^n$ is an encoding scheme (enc, dec) such that $\mathsf{enc} : \{0,1\}^n \to \{0,1\}^k$ and $\mathsf{dec}(\mathsf{enc}(s)) = s$ for all $s \in Sup(\mathcal{S})$.*

Definition 3 (k-Compressible)**.** *A source $\mathcal{S}$ is said to be k-compressible (or compressible to k bits) if there exists a $(\mathcal{S}, k)$-compression and incompressible to k bits otherwise.*

By this definition of an $(\mathcal{S}, k)$-compression scheme it is necessary that k is at least the max entropy of $\mathcal{S}$, i.e. $k \geq \log_2 |\mathrm{Sup}(\mathcal{S})|$, because it must uniquely encode every string in the support. A lossy compression scheme compromises on the probability of recovery, but may be able to compress to fewer bits than the max entropy.

Definition 4. *A δ-lossy $(\mathcal{S}, k)$-compression scheme is an encoding scheme (enc, dec) where $\mathsf{enc} : \{0,1\}^n \to \{0,1\}^k$ and $Pr_{s \xleftarrow{\mathrm{R}} \mathcal{S}}[\mathsf{dec}(\mathsf{enc}(s)) = s] = 1 - \delta$. $\mathcal{S}$ is said to be δ-lossy compressible to k bits if and only if such an encoding scheme exists and δ-lossy incompressible otherwise.*

Randomized compression In randomized compression schemes, enc and dec use an additional entropy source to aid in compression. In the most general case enc and dec may simply run independent randomized algorithms. More typically, they will share information about a *seed*, either communicated through the encoding output itself or pre-shared, which we may assume is a uniformly random string $\rho \xleftarrow{\mathrm{R}} \mathbb{U}_r$ ($\mathbb{U}_r$ denotes the uniform distribution over $\{0,1\}^r$). Following the terminology of [27] we distinguish *weak* and *strong* randomized compression (similar to weak vs strong randomness extractors). Strong compression stores the seed separately from the compressed data, and is equivalent to compression with shared randomness. Weak compression may entangle the seed and the data in the code in such a way that the decoder can still recover both the data and the seed.

Definition 5 (Weak compression). *A δ-lossy $(\mathcal{S}, r, k)$-weak compression for a source $\mathcal{S}$ over $\{0,1\}^n$ is an encoding scheme $(\mathsf{enc}, \mathsf{dec})$ such that $\mathsf{enc} : \{0,1\}^r \times \{0,1\}^n \to \{0,1\}^{r+k}$ and $Pr_{\rho \xleftarrow{\mathrm{R}} \mathbb{U}_r, s \xleftarrow{\mathrm{R}} \mathcal{S}}[\mathsf{dec}(\mathsf{enc}(r,s)) = (r,s)] = 1 - \delta$.*

Definition 6 (Strong compression). *A δ-lossy $(\mathcal{S}, r, k)$-strong compression for a source $\mathcal{S}$ over $\{0,1\}^n$ is an encoding scheme $(\mathsf{enc}, \mathsf{dec})$ where $\mathsf{enc} : \{0,1\}^r \times \{0,1\}^n \to \{0,1\}^k$ and $\mathsf{dec} : \{0,1\}^r \times \{0,1\}^k \to \{0,1\}^n$ such that $Pr_{\rho \xleftarrow{\mathrm{R}} \mathbb{U}_r, s \xleftarrow{\mathrm{R}} \mathcal{S}}[\mathsf{dec}(\rho, \mathsf{enc}(\rho, s)) = s] = 1 - \delta$.*

A lossy strong compression scheme can be made lossless for a small decrease in compression [49]. Even strong compression schemes cannot compress a uniformly random data source [22].

Fact 1. *If there exists a δ-lossy $(\mathcal{S}, r, k)$-strong compression for a source $\mathcal{S}$ then there exists an $(\mathcal{S}, r, k + \delta(n - k) + 1)$-strong compression of $\mathcal{S}$ with no loss.*

Fact 2. *The uniform distribution U_n over $\{0,1\}^n$ is incompressible to $n - 1$ bits and δ-lossy incompressible to $n - \log(1/\delta) - 1$. Likewise, there does not exist a δ-lossy $(U_n, r, n - \log(1/\delta) - 1)$-strong compression scheme.*

2.2 Proofs of Retrievability

A proof-of-retrievability (PoR) is an interactive proof system in which a verifier sends a file F to a prover, retains a compact verification key, and later obtains a compact proof that prover can retrieve F intact. The compactness requirement excludes trivial solutions, such as sending the full file F back to the verifier or requiring the verifier to retain F. They were introduced in [30] and further developed in [15, 23, 48]. The related (weaker) concept of provable-data-possession (PDP) was independently introduced in [8]. Abstractly, these constructions involve two layers of encodings. First an erasure code is applied to the file F to generate $\tilde{F}$, which guarantees that F can be decoded from adversarial deletion of an ϵ fraction of $\tilde{F}$. Next, the file $\tilde{F}$ is transformed using a secret-key encoding (essentially a type of authentication code), and the verifier retains the key. The second encoding can be checked through a constant size challenge-response protocol, which the prover will fail if it cannot recover at least a $1 - \epsilon$ fraction of $\tilde{F}$. An important security property of PoR is that the verifier can extract and recover the file F through sufficiently many successful challenge-response queries to the prover.

While a standard PoR requires a secret key to verify, there are also constructions that allow for outsourced verification where anyone can verify without knowledge of the secret [30, 48]. We refer to this as a *public-key* PoR to distinguish it from a *private-key* PoR that requires a private key for verification. Shacham and Waters [48] construct an efficient public-key PoR (from CDH in bilinear groups) using the homomorphic properties of BLS signatures. Their construction was generalized to rely on a broader class of homomorphic linear authenticators [9]. This property has been described as "public verifiability", however these proofs still rely on a private-key client to privately preprocess the file. If the prover and client collude then the proof does not demonstrate anything meaningful to a third party observer.

A Merkle commitment (i.e. a Merkle tree over the blocks of the file) can be be used to construct a simple, albeit less efficent, publicly verifiable PoR [30]. The verifier need only retain the Merkle commitment root. To verify that the prover is still storing a $(1 - \epsilon)$ fraction of the committed file blocks it queries for a randomly selected constant number of blocks. The prover then responds with the blocks and Merkle inclusion proofs for each. This public PoR is keyless, which is an important difference from the public PoR of Shacham and Waters. For example,

it can be used to give a publicly verifiable proof-of-retrievability of a committed file, where the even the prover itself sets up the commitment. We present this as a standalone primitive and describe a generalized construction using vector commitments.

2.3 Vector Commitments

A vector commitment (VC) [17,35] is a cryptographic commitment to an ordered sequence of m values $(x_1, ..., x_m)$ that admits succinct openings at specific positions (e.g., prove that x_i is the i-th committed message). For security, VCs are required to satisfy *position binding*, which states that an adversary should not be able to open a commitment to two different values at the same position. Moreover, VCs are required to be *concise*, i.e. the size of the commitment string and of its openings is independent of the vector length. Usually VCs are also required to be hiding, meaning that opening at several positions does not leak any information about the committed values at other positions. A Merkle tree is an example of a simple vector commitment that is binding and concise but not hiding. The main drawback of Merkle commitments are that proofs are of size $\lambda \log_2 m$ bits where m is the length of the vector and λ is the length of the hash output (e.g. $\lambda = 256$ for collision resistance).

Catalano and Fiore [17] gave several constructions of vector commitments achieving both hiding and constant size openings, one from RSA and the other from CDH over bilinear groups. These were further improved upon in [34]. The main drawback of these constructions are the size of the public parameters, which are linear in the length of the vectors. As noted in [17], the RSA-based vector commitment can be modified to reduce the size of the public parameters, but at the expense of a significant increase in verification work. In Section **??** we point out a very useful batching property of the RSA-vector commitment from [17], namely that the simultaneous opening of several indices can be batched into a single constant size proof.

Cryptographic accumulators [10, 13] give constant size commitments to a set of elements with constant size inclusion proofs of a single element, however they do not yield a vector commitment (i.e. commitment to an ordered set) with efficient inclusion proof at a specified index. This is due the fact that they are not naturally binding to an element at a particular index—the natural idea of prepending an index to each element inserted does not preclude inserting more than one element with the same prepended index. In Section **??** we describe a new vector commitment construction based on RSA accumulators [16] that has constant size public parameters and $O(\lambda \log m)$ size openings ($\lambda \approx 1024 - 2048$ based on current beliefs for strong RSA security). The opening of a single vector component is no more compact that a Merkle commitment opening, however the openings of several n-bit vector components can be batched in a way that achieves an amortized $O(min(n, \lambda) \cdot \log m)$ bits per message, which is concretely smaller than a Merkle opening when $n << 256$.

Vector commitment syntax   We provide a redacted syntax for vector commitments (omitting functionality for updating commitments) and allowing for batched openings.

1. $\mathsf{VC.Setup}(1^\lambda, n, \mathcal{M}) \to pp$ Given security parameter λ, length n of the vector, and message space of vector components $\mathcal{M}$, output public parameters pp, which are implicit inputs to all the following algorithms.

2. $\mathsf{VC.Com}(\vec{m}) \to \tau, c$ Given an input $\vec{m} = (m_1, ..., m_n)$ output a commitment c and advice τ.

3. $\mathsf{VC.Open}(\vec{m}, c, S, \tau) \to \Lambda_S$ On input $\vec{m}$, a vector of indices S, the commitment c, and advice τ output an opening Λ_S that proves m_i is the ith committed element of c for all $S[i]$ (where $S[i]$ denotes the ith element of S).

4. $\mathsf{VC.Verify}(c, m'_1, ..., m'_q, S, \Lambda_S) \to 0/1$ On input commitment c, a vector S of indices in $[n]$, and an opening proof Λ_S output 1 (accept) if and only if Λ_S is a valid proof that c is a commitment to a vector $\vec{m}$ such that $m_{S[j]} = m'_j$ for $j \in [q]$. If $S = \emptyset$ then Λ should be a normal opening, i.e. proof that c is a commitment to m s.t. $m_i = m'_i$ for all i (and $q = n$).

Binding commitments The main security property of vector commitments (of interest in the present work) is position binding. The security game augments the standard binding commitment game

Definition 7. *A vector commitment scheme* VC *is position binding if for all* $O(poly(\lambda, n))$-*time adversaries* $\mathcal{A}$ *and for all* $S', S \subseteq [n]$ *with* $|S| = q$ *and* $|S'| = q'$ *and* $i \in [q], j \in [q']$ *the following probability is at most negligible in* λ:

$$
Pr \left[\begin{array}{l} \mathsf{VC.Verify}(c, \vec{m}, S, \Lambda) = 1 \ \wedge \ \mathsf{VC.Verify}(c, \vec{m}', S', \Lambda') = 1 \\ \wedge \ S[i] = S'[j] \wedge \ m_i \neq m'_j \end{array} \middle| \begin{array}{l} pp \leftarrow \mathsf{VC.Setup}(1^\lambda, n, \mathcal{M}) \\ (c, \vec{m}, \vec{m}', \Lambda, \Lambda') \leftarrow \mathcal{A}(pp) \end{array} \right]
$$

2.4   Proof of Retrievable Commitment

We describe here a primitive that we call a *proof of retrievable commitment* (PoRC), or alternatively a *keyless* PoR, which allows a prover to give a compact proof of retrievability for a committed value. Similar to a public PoR, the proof verification algorithm does not take a private key. However, it additionally does not involve any private-key preprocessing of the file. Therefore, unlike public-key PoRs where the verification may only be *outsourceable*, PoRs are truly publicly verifiable: a PoRC is a standard PoR to any verifier that is first given the opening of the commitment (to a particular file) and retains the commitment as a tag. Indeed the concepts of a PoRC and keyless PoR are roughly equivalent, as any keyless PoR must involve a keyless data extractor, implying that it can be used as a binding commitment scheme (where opening is through extraction).[4] A PoRC is also similar to a succinct proof of knowledge (PoK), or SNARK [11], of a committed value. The difference lies still in the type of extraction, which unlike *knowledge extraction* does not require reading the prover's internal state.

The notion of a PoRC was already implicit in the literature on proofs of retrievability, a simple example being the PoR from Merkle commitments suggested by [30], although a formal presentation and proof of security was not given therein. We present a more general construction of PoRC from any vector commitment scheme and provide a formal proof of security.[5]

---

[4]This has also been referred to as a PoR with public extraction in [48].

[5]Our proof is a straightforward coupon collector's argument, following the same proof technique from [48] and [23]. The reason the proof is non-trivial to begin with arises from the fact that all challenges are given to the prover in parallel (rather than sequentially) to amplify soundness. This is more broadly connected to *direct product theorems* in hardness amplification. In fact, Lemma 1 follows from a more general and previously known result (Lemma 7 of [23]), however we still include our own analysis in the special setting of PoRC from vector commitments, both for completeness of presentation and the benefit of the reader.

2.4.1 PoRC Syntax and Definitions

Our syntax for a PoRC with a message space $\mathcal{M} \subseteq \{0,1\}^n$ is as follows.

1. PoRC.Setup$(1^\lambda, \mathcal{M}) \to pp, \mathcal{M}$ on input security parameter λ and message space $\mathcal{M}$ output public parameters pp.

2. PoRC.Commit$(pp, m) \to com, \Lambda, \alpha$ on input pp and a message $m \in \mathcal{M}$ outputs a binding commitment com to the message m along with an opening hint Λ and additional advice α for the prover.

3. PoRC.Open$(pp, com, \Lambda) \to m$ opens the commitment com to a message $m \in \mathcal{M} \cup \perp$ given the opening hint Λ. The output is $\perp$ if and only if Λ is an invalid opening.

4. PoRC.Prove$(pp, \alpha, com, c) \to \pi$ is a proof generation algorithm that takes pp, the storage advice α, the commitment com, a challenge c (from either the verifier or possibly a public source) and outputs a proof π.

5. PoRC.Verify$(pp, com, c, \pi) \to 0/1$ receives pp, the commitment com, challenge c, and proof π and outputs either 1 (accept) or 0 (reject).

PoRC protocol We define the PoRC interactive protocol between a prover $\mathcal{P}$ and verifier $\mathcal{V}$. The PoRC interactive protocol as we define it here is public coin, but could be expanded to allow for private coin protocols as well. First $\mathcal{P}$ runs PoRC.Commit(pp, m), sends the output com to the verifier, and stores both the opening hint Λ and additional advice α. Next $\mathcal{V}$ samples a random challenge c and sends this to $\mathcal{P}$. Finally, $\mathcal{P}$ runs PoRC.Prove(pp, α, com, c) and sends the output π to $\mathcal{V}$, who runs PoRC.Verify(pp, com, c, π) and outputs its decision $0/1$. Note that PoRC.Open is not used in the protocol, but its existence (satisfying a message binding property) is necessary for security. Furthermore, a verifier may use PoRC.Open to obtain an opening of the commitment at any point, or alternatively its description (e.g. in order to verify a zero-knowledge proof of some property of the message).

Correctness For a PoRC to be correct we require the commitment com to be compact, i.e. logarithmic in the length of m. Furthermore, in the PoRC interactive protocol with an honest prover and verifier (for any $m \in \mathcal{M}$) should output 1 (accept) with probability 1. (We omit further formalism as this is standard).

Soundness The PoRC soundness property is more subtle and has two components: commitment binding and retrievability. These properties are intertwined as together they say that for any *admissible* prover (loosely one that is sufficiently successful) there exists an extraction process that extracts a message m unique to the commitment com.

The first requirement is that PoRC.Commit/PoRC.Open is a binding commitment scheme in the standard sense:

Definition 8. *A PoRC scheme is binding if for all $O(poly(\lambda))$ adversaries $\mathcal{A}$ the probability over $pp \leftarrow$ PoRC.Setup$(\lambda, \mathcal{M})$ and $(\Lambda, \Lambda', c^*) \leftarrow \mathcal{A}(pp)$ and $m \leftarrow$ PoRC.Open(pp, c^*, Λ) and $m' \leftarrow$ PoRC.Open(pp, c^*, Λ') that $m' \neq m \neq \perp$ is negligible in λ.*

Next we define retrievability. Intuitively, the desired property is that any adversary who is a able to produce a commitment *com*, open this commitment to a value m (e.g. for a client), and then succeed with non-negligible probability in the PoRC protocol with a public verifier must still be able to retrieve the value m, i.e. there is an extractor that can run the prover on public verifier challenges in order to uniquely extract the same value m. We define a game played between an adversary $\mathcal{A} = (A_1, A_2)$ and a challenger which essentially just runs the public-coin PoRC protocol. The game is simpler than the PoR games in [23, 48] because of our restriction to a public coin verifier. The game is parametrized by input $m \in \mathcal{M}$ and public parameters pp and runs as follows:

1. A_1 on input pp and m outputs a message com^* (adversarial commitment) along with α^*, Λ^*. It passes α^* to A_2 and sends com^*, Λ^* to the challenger. The challenger runs $m^* \leftarrow$ PoRC.Open(pp, com^*, Λ). If $m^* \neq m$ then the challenger outputs 0 and terminates the game.

2. The challenger samples a challenge c uniformly at random from the domain of challenges and sends this to A_2.

3. A_2 (running on c, α^*, and com^*) outputs an adversarial proof π^*.

4. The challenger computes $b \leftarrow$ PoRC.Verify(pp, com^*, c, π^*) and outputs b.

For any $m \in \mathcal{M}$ we denote the challenger's output in the game as a probability distribution PoRC-Exp$(\mathcal{A}, m, pp)$ over the randomness of $\mathcal{A}$, the challenger's queries, and $pp \leftarrow$ PoRC.Setup$(\lambda, \mathcal{M})$.[6]

Definition 9. *A PoRC adversary $\mathcal{A}$ is ϵ-admissible on input m if and only if A_1 outputs com^* such that PoRC.Open$(pp, com^*, \Lambda) \neq \perp$ with probability $1 - negl(\lambda)$ and PoRC-Exp$(\mathcal{A}, m, pp) = 1$ with probability at least ϵ.*

We define a PoRC extractor $\mathcal{E}$ (akin to the extractor in a proof of knowledge) as an oracle algorithm that is given the output com^* of the PoRC adversary $\mathcal{A}$ and simulate the interaction between the challenger and A_2. An important modification is that the extractor can rewind A_2 and reissue fresh challenges (for further formalism on extractor machines that can rewind adversaries see [25, 37]). The key difference between our extractor and a standard knowledge extractor is that it cannot read the internal state or inputs of the adversary in the game, only the power to rewind its randomness. More specifically, in each run of the extractor playing the game PoRC-Exp$(\mathcal{A}, m, pp)$ with $\mathcal{A}$, it first obtains a commitment c^* and then repeatedly queries A_2 (possibly rewinding A_2 at some points), until it finally outputs either a message $m^* \in \mathcal{M}$ or $\perp$ (indicating failure). Let $m^* \leftarrow \mathcal{E}^{\mathcal{A}(m)}(pp)$ denote the output distribution of this experiment.

Definition 10. *A PoRC is ϵ-sound if it is binding and additionally for every λ there exists a PoRC extractor $\mathcal{E}$ that runs in time $poly(\lambda, n, 1/\epsilon)$ such that for any $m \in \mathcal{M}$ and any adversary $\mathcal{A}$ that is ϵ-admissible on m the probability over $pp \leftarrow$ PoRC.Setup$(\lambda, \mathcal{M})$ and $m^* \leftarrow \mathcal{E}^{\mathcal{A}(m)}(pp)$ that $m^* \neq m$ is at most $negl(\lambda)$.*

---

[6]The adversary in the game is non-adaptive, which at first glance may appear to weaken the adversary. Note that the challenger in the game is entirely public-coin. The adversary can simulate any polynomial number of adaptive rounds of the game in his head before engaging in a final real round with the challenger.

Definition 10 captures that any adversary who passes the PoRC protocol with non-negligible probability at least ϵ must be able to retrieve the underlying data by running the extractor in time $O(1/\text{poly}(\epsilon))$ with its own code.[7] If on the other hand the adversary does not pass the PoRC with probability at least ϵ then the soundness error of the protocol with this adversary is ϵ in the standard sense (i.e. it is caught with probability at least $1 - \epsilon$).

Extractor runtime Extractors always have a larger runtime complexity than the original adversary $\mathcal{A}$. Extractors are considered *stronger* the smaller this overhead. In our case A_1 first generates a commitment that is then given to A_1 (we do not count the runtime of A_1 on pp as part of the extractor's runtime). We say that the extractor has *constant blowup* if the runtime is only a constant factor larger than A_2. It turns out that the parallel runtime of the PoRC extractor will be highly relevant to the security analysis of PoRep schemes. We say the extractor has *constant PRAM blowup* if the parallel runtime is only a constant factor larger than the parallel runtime of A_2.

Definition 11. *A PoRC is PRAM strongly ϵ-sound if it is ϵ-sound and additionally the extractor has constant PRAM blowup.*

Partial retrievability Sometimes proofs of retrievability can be useful even if they only prove partial recovery, e.g. that a constant fraction of the bits of the message can be retrieved. In fact, all known constructions of PoR effectively first prove partial recovery, which is amplified to a proof of full recovery through erasure coding. The proof of partial recovery itself is more efficient because it does not require blowing up the size of the message. Even if partial recovery is not the end goal it is useful to abstract this as a standalone primitive that can be used as a subcomponent of cryptographic protocol (in our case PoReps).

We define a δ-PoRC as a proof that a δ fraction of a committed message can be retrieved. We generalize this even further to a $(\delta, \mathcal{C})$-PoRC where $\mathcal{C}$ is any *set cover* of the committed message m, each set consisting of bit indices of m (assume w.l.o.g. that m is represented as a bit vector). This is useful when it is important to ensure that the missing fraction of bits isn't completely arbitrary, for example that at least a δ fraction of complete message blocks are retrievable. A δ-PoRC is equivalent to the special case where the set cover consists of disjoint singleton sets, each containing a single bit of the message.

The security game and extractor for a $(\delta, \mathcal{C})$-PoRC is unchanged from the definition of PoRC and only the definition of ϵ-soundness needs to be modified. For any pair of equal length messages $|m| = |m'| = n$ and index subset $S \subseteq [n]$ let $m \equiv_S m'$ denote that $m_i = m'_i$ for all $i \in S$.

Definition 12. *A $(\delta, \mathcal{C})$ PoRC is ϵ-sound if it is binding and additionally for some[8] $\lambda_{\epsilon,\delta} \in O(\log \epsilon / \log \delta)$ and every $\lambda > \lambda_{\epsilon,\delta}$ there exists an extractor $\mathcal{E}$ that runs in time $\text{poly}(\lambda, n, 1/\epsilon)$ such that for any $m \in \mathcal{M}$ and any adversary $\mathcal{A}$ that is ϵ-admissible on m the probability over $pp \leftarrow \textsf{PoRC.Setup}(\lambda, \mathcal{M})$ and $m^* \leftarrow \mathcal{E}^{\mathcal{A}(m)}(pp)$ that $m \equiv_S m^*$ for at least a δ fraction of the sets $S \in \mathcal{C}$ is greater than $1 - \textsf{negl}(\lambda)$. It is PRAM strongly ϵ-sound if $\mathcal{E}$ has constant PRAM blowup.*

---

[7]Due to the fact that the extraction algorithm is public, this is stronger than standard knowledge extraction, where only the adversary who generated the adversarial code in the first place necessarily can extract the witness. For instance, the interactive adversary in the protocol may have been given obfuscated code that hides the witness.

[8]The definition would still be technically equivalent in an asymptotic sense if it read "for every λ" without further requirement, however the rewording helps to clarify that in general we cannot hope the extractor will succeed for $\epsilon \leq \delta^\lambda$.

2.4.2 PoRC from Vector Commitments

We describe a construction of a $(\delta, \mathcal{C})$-PoRC for blocked messages of a fixed length consisting of n blocks from a message space $\mathcal{M}$, and any set cover $\mathcal{C}$ consisting of unions of at most $O(\log n)$ blocks. For applications with variable length commitments we can run the setup for each $n = 2^k$ up to a maximum size. The construction also easily generalizes to set covers consisting of sets that are unions of at most $O(\text{polylog}(n))$ bit indices, regardless of the block size. We use a vector commitment scheme $\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Com}, \mathsf{VC.Open}, \mathsf{VC.Verify})$ as a black box.

Construction VC-PoRC

1. $\mathsf{PoRC.Setup}(1^\lambda)$: Run $\mathsf{VC.Setup}(1^\lambda, n, \mathcal{M}) \to pp_{vc}$. Output pp.

2. $\mathsf{PoRC.Commit}(pp, m)$: Parse m as a vector $\vec{m} = (m_1, ..., m_n)$ where $m_i \in \mathcal{M}$. Run $\mathsf{VC.Com}(\vec{m}) \to \tau, com$, and output τ, com.

3. $\mathsf{PoRC.Prove}(pp, \tau, com, c)$: Let $\mathcal{C} = \{S_1, ..., S_k\}$ where each $S_i \subset [n]$. Parse c as integers $c_1, ..., c_\lambda \in [k]$. Let $S = \bigcup_{i=1}^\lambda S_{c_i}$. Run $\mathsf{VC.Open}(\vec{m}, c, S, \tau) \to \Lambda$. Output the proof π consisting of all $\{m_j\}$ for $j \in S$ and Λ.

4. $\mathsf{PoRC.Verify}(pp, com, c, \pi)$: On $c = c_1, ..., c_\lambda \in [k]$ define $S = \bigcup_{i=1}^\lambda S_{c_i}$. Parse $\pi = (m'_1, ..., m'_\ell, \Lambda)$ and output 0 (reject) if $|S| \neq \ell$. Run $\mathsf{VC.Verify}(com, m'_1, ..., m'_\ell, S, \Lambda)$ (given input S as a sorted ordered list) and output the result.

Proof size Let $s(n, k)$ denote an upper bound on the size of proofs output by $\mathsf{VC.Open}$ for vectors of length n over the message space $\mathcal{M}$ and index subset of size k. Let k_{max} denote the largest set in $\mathcal{C}$. The total proof size output by $\mathsf{PoRC.Prove}$ is at most $s(n, \lambda k_{max}) + \lambda k_{max} log(|\mathcal{M}|)$, which by definition of VC is asymptotically $O(\lambda \log^2 n)$. For instantiations of VC with constant size openings, i.e. $s(n, k) \in O(\lambda k)$, the proof size is $O(\lambda^2 \log n)$. If VC supports batching the size is brought down to $O(\lambda \log n)$, which is the size of the λ sampled sets (i.e. the vector commitment opening proof is no longer the bottleneck).

Correctness As explained above, the proof size is compact, i.e. at most $O(\lambda \log^2 n)$. An honest prover who stores the input message m is able to pass verification by correctness of the vector commitment scheme VC.

Soundness We show that there is an extractor $\mathcal{E}$ such that $\mathcal{E}^{\mathcal{A}(m)}(pp)$ outputs the file m with overwhelming probability for any adversary $\mathcal{A}$ that is ϵ-admissible on m, where $\epsilon > \delta^\lambda$ is non-negligible.

Theorem 1. *VC-PoRC with security parameter λ is an ϵ-sound $(\delta, \mathcal{C})$-PoRC for any $\epsilon > 0$ such that $\epsilon - \delta^\lambda > \mathsf{negl}(\lambda)$.*

Proof. We define $\mathcal{E}_\mathcal{A}$ with $\mathcal{A} = (A_1, A_2)$ is given pp from $\mathsf{PoRC.Setup}(1^\lambda)$ (sampled in the game $\mathsf{PoRC\text{-}Exp}(\mathcal{A}, \lambda)$), and the index set cover $\mathcal{C} = \{S_1, ..., S_k\}$ and runs as follows.

1. Query A_1 to get com^*. Initialize a round counter $r = 0$. Initialize a set $D = \emptyset$ and initialize a length n vector $\vec{v} = (0, ..., 0)$ of 0s.

2. Increment the counter $r := r + 1$. Sample a random challenge vector $c = (c_1, ..., c_\lambda) \in [k]^\lambda$ and query A_2 on c and record the response π_c.

3. Run $b \leftarrow \mathsf{PoRC.Verify}(pp, com^*, c, \pi_c)$. If $b = 0$ do nothing. If $b = 1$, let $S = \bigcup_{i=1}^\lambda S_{c_i}$ and parse $\pi_c = (m'_1, ..., m'_\ell, \Lambda)$ where $\ell = |S|$ (if π_c could not be parsed this way it would have been rejected). Write S as a sorted list of its elements and for any $j \in [\ell]$ let $S[j]$ denote the index at the jth position of the sorted list. If any $c_i \notin D$ (for $1 \le i \le \lambda$), then update $D := D \cup \{c_i\}$. For each $j \in [\ell]$ update $v_{S[j]} = m'_j$.

4. If $r = \frac{2(\lambda+1)\delta n}{(\epsilon - \delta^\lambda)^2}$ or if $|D| \ge \delta k$ then terminate and output $m^* = \vec{v}$. Otherwise rewind A_2 and repeat from Step 2.

The extractor's goal is to obtain components of m for all indices covered by a δ fraction of index sets in $\mathcal{C}$. Step 3 runs the adversary on a fresh challenge to open com^* on all indices of λ sampled sets from $\mathcal{C}$. If the adversary passes the challenge (verification succeeds) in Step 3, then by the security of the commitment scheme the extractor obtains the components of m on all these queried indices. The extractor checks off any newly covered index sets and terminates if it has already covered at least a δ fraction. Otherwise it repeats (rewinding the adversary to use the same randomness) until the max round is reached. Finally, note that although in the description of $\mathcal{E}_\mathcal{A}$ runs for r sequential rounds, observe that all the queries to A_2 (Step 2) are independent of state and thus can be run in parallel. A_2 is rewound for each query, therefore querying r copies of A_2 in parallel achieves the same result. The bookkeeping in Step 3 can be run in parallel on the outputs of all these queries.

Lemma 1 proves that the $(\delta, \mathcal{C})$-PoRC from vector commitments is ϵ-sound for any $\epsilon > \delta^\lambda + \mathsf{negl}(\lambda)$. In light of the above remark, the construction is PRAM strongly ϵ-sound with only constant *additive* PRAM blowup. $\qquad\square$

Lemma 1. *Assuming* VC *is a position binding vector commitment scheme, then for any ϵ-admissible $\mathcal{A}$ where $\epsilon > \delta^\lambda$ is non-negligible, the extractor $\mathcal{E}_\mathcal{A}$ outputs m^* such that $m^* \equiv_S m$ for at least a δ fraction of index sets $S \in \mathcal{C}$ except with probability $2^{-\lambda}$.*

Proof. First recall that an ϵ-admissible adversary will cause the challenger to output 1 in Step 1 except with negligible probability.

From here the proof proceeds in two parts. First, we say that $\mathcal{A} = (A_1, A_2)$ is *well-behaved* if whenever A_1 passes Step 1 of the game $\mathsf{PoRC\text{-}Exp}(\mathcal{A}, \lambda))$ (i.e. presents com^* and $m = (m_1, ..., m_n)$ to the challenger with opening Λ such that $\mathsf{VC.Verify}(com^*, m_1, ..., m_n, \Lambda) = 1$) and A_2 responds to some query c with a proof π_c such that $\mathsf{PoRC.Verify}(pp, com^*, c, \pi_c) = 1$ then $\pi_c = (m'_1, ..., m'_\ell, \Lambda')$ and $m'_j = m_{S[j]}$ for all $j \in [\ell]$, where $S = \bigcup_{i=1}^\lambda S_{c_i}$ (the indices covered by the query c). We claim that $\mathcal{A}$ is well-behaved except with probability $\mathsf{negl}(\lambda)$.

Indeed if the PoRC verification passes on π_c then $\mathsf{VC.Verify}(com^*, m'_1, ..., m'_\ell, S, \Lambda') = 1$. If $m'_j \ne m_{S[j]}$ on some j then $com^*, m', \Lambda', m, \Lambda$ is an opening collision of the vector commitment scheme. If this happens with non-negligible probability (conditioned on A_1 passing Step 1), then because by hypothesis A_1 passes Step 1 at least with non-negligible probability ϵ, we can use $\mathcal{A}$ to break the position binding of VC with non-negligible probability, a contradiction.

For the second part we assume that $\mathcal{A}$ is well-behaved. Queries c for which A_2 passes verification are "good" queries, and all other queries are "bad" queries. Consider any iteration of the extractor where $D \subset [n]$ and $|D| < \delta k$. If the extractor asks a "good" query $c = (c_1, ..., c_\lambda)$

15

such that $c_w \notin D$ for some $w \in \lambda$, then in Step 3 it adds c_w to D increasing the size by (at least) 1. Furthermore, because $\mathcal{A}$ is well-behaved the extractor updates $v_{S[j]} = m'_j = m_{S[j]}$ for each index $S[j], j \in \ell$ covered by the query. Thus at the end of each round, for any $i \in D$ the vector v is identical to m on all indices in S_i, i.e. $v \equiv_{S_i} m$. If the extractor asks a "bad" query or a query that does not hit any new sets, i.e. $c_w \in D$ for all $w \in [\lambda]$, then the size of D does not increase in this round. The query is "bad" with at most probability $1 - \epsilon$ and since each c_w is uniformly sampled from $[k]$ the latter occurs with probability at most δ^λ. Thus, by a union bound, in each round the size of D increases by 1 with probability at least $1 - (1 - \epsilon + \delta^\lambda) = \epsilon - \delta^\lambda$.

Let r denote the total number of rounds, and let X_i for $1 \le i \le r$ be an indicator random variable such that $X_i = 1$ if either $|D|$ increases in the ith round or $|D| \ge \delta k$ by the ith round. Above we established that $Pr[X_i = 1] \ge p = \epsilon - \delta^\lambda$. The size of D after r rounds is at least $min(\sum X_i, \delta k)$. Thus it suffices to establish that $\sum X_i \ge \delta k$ with probability at least $1 - 2^{-\lambda}$. The variables X_i are not independent so we cannot immediately apply the Chernoff-Hoeffding bound. However due to the fact that $Pr[X_i] \ge p$ conditioned on any values of $X_1, ..., X_{i-1}$, if we define independent 0/1 variables $Z_1, ..., Z_r$ such that $Pr[Z_i = 1] = p$ for all i, then it follows that $Pr[\sum_i X_i \le \delta k] \le Pr[\sum_i Z_i \le \delta k]$. Finally, setting $r \ge (2\delta k/p)$ and the Chernoff-Hoeffding bound then implies: $Pr[\sum_i Z_i \le \delta k] \le Pr[\sum_i Z_i \le rp/2] < 2e^{-rp^2/2}$. Finally we can choose r such that $rp^2/2 \ge \lambda + 1$. Finally, to satisfy both conditions we set $r \ge max(2\delta k/p, 2(\lambda + 1)/p^2)$, e.g. $r = 2\delta k(\lambda + 1)/p^2 = 2\delta k(\lambda + 1)/(\epsilon - \delta^\lambda)^2$.

$\square$

2.5 Verifiable Delay Encodings

A *verifiable delay encoding* (VDE) is an encoding that is slow to compute yet fast to decode. The encoding requires non-parallelizable sequential work to evaluate and therefore in theory cannot be computed in shorter than some minimum wall-clock time. A VDE is a special case of a *verifiable delay function* recently introduced in [14]. Practical (heuristic) examples of VDEs include Sloth [31], MiMC [4], and a special class of permutation polynomials [14].

Formally, a VDE is a tuple of three algorithms VDE = VDE.Setup, VDE.Enc, VDE.Dec defined as follows.

1. VDE.Setup$(t, \lambda) \rightarrow pp$ is given security parameter λ and delay parameter t produce public parameters **pp**. By convention, the public parameters also specify an input space $\mathcal{X}$ and a code space $\mathcal{Y}$. We assume that $\mathcal{X}$ is efficiently samplable. VDE.Setup might need secret randomness, leading to a scheme requiring a trusted setup.

2. VDE.Enc$(pp, x) \rightarrow y$ takes an input $x \in \mathcal{X}$ and produces an output $y \in \mathcal{Y}$.

3. VDE.Dec$(pp, y) \rightarrow x$ takes an input $y \in \mathcal{Y}$ and produces an output $x \in \mathcal{X}$.

Correctness For all pp generated by VDE.Setup(λ, t) and all $x \in \mathcal{X}$, algorithm VDE.Enc(pp, x) must run in parallel time t with $poly(\log(t), \lambda)$ processors, and VDE.Dec$(pp, \text{VDE.Enc}(pp, x)) = x$ with probability 1.

Parallelism The practical significance of allowing $poly(\log(t), \lambda)$ parallelism in VDE.Enc is that any implementation must have this much parallelism in order to evaluate within the delay time t. The security property of a VDE will bound the success probability of an adversary

running in less than time t. If the construction demands VDE.Enc to use a significant amount of parallelism to complete in a time t for which this sequentiality bound holds (i.e. the adversary cannot gain further speedup) then it may only be useful for applications where the "honest" evaluators have this much parallelism available. Constructions that do not require any parallelism to evaluate VDE.Enc in the optimal number of sequential steps are obviously superior. However, it is unlikely that such constructions exist (without trusted hardware). In fact, it is necessary that $\mathcal{Y} > \text{poly}(t)$ (otherwise the output can be guessed and checked quickly using VDE.Dec) and hence the challenge inputs have size $\text{poly}\log(t)$. Therefore, at a minimum we must allow up to $\text{poly}\log(t)$ parallelism. In practice, much of this parallelism may already be taken advantage of by the hardware (e.g. word sizes on modern processors and/or SIMD instructions).

2.5.1 VDE security

The main security property of a VDE is σ-sequentiality, which characterizes that no adversary running in parallel time significantly less than t is able to compute VDE.Enc on a *random* input in $\mathcal{X}$, even given preprocessing advice.[9] This captures an unpredictability property given time constraints on the adversary.

Definition 13 (sequentiality). *For a function $\sigma(t)$ a VDE is σ-sequential if for any pair of randomized algorithms $\mathcal{A}_0$, which runs in total time $O(poly(t, \lambda))$, and $\mathcal{A}_1$, which runs in parallel time $t - \sigma(t)$ on at most $O(poly(t))$ processors, the following probability distribution over $pp \leftarrow$ VDE.Setup(t, λ) is negligible:*

$$ Pr\left[\begin{matrix} y \leftarrow \mathcal{A}_1(\alpha, \mathbf{pp}, x) \\ \wedge\ y = \textsf{VDE.Enc}(x) \end{matrix} \middle| \begin{matrix} x \xleftarrow{\text{R}} \mathcal{X} \\ \alpha \leftarrow \mathcal{A}_0(\lambda, \mathbf{pp}, t) \end{matrix} \right] < \mathsf{negl}(\lambda) $$

A second (stronger) property concerns the pseudorandomness of the output. Clearly the output of VDE.Enc on an input x is not indistinguishable from random as we can run VDE.Dec to recover the input, however we may want the property that $\mathsf{Encode}(pp, \cdot)$ is indistinguishable from a random permutation (if $\mathcal{X} = \mathcal{Y}$) for pp sampled from the setup, similar to modeling block ciphers as an *ideal cipher*. We formalize the ideal delay encoding object (for the special case of a permutation) as an oracle in the following definition.

Definition 14. *An ideal delay permutation (IDP) is a family of oracles $\{\mathcal{O}_{IDP}^{(t)}\}$ that implement a random permutation Π and respond to two types of queries. On a query $(q, 0)$ the oracle $\mathcal{O}_{IDP}^{(t)}$ internally simulates t sequential queries to Π^{-1} and then outputs $\Pi(q)$. On a query $(q, 1)$ it outputs $\Pi^{-1}(q)$.*

Indifferentiability on random inputs One would hope to instantiate an IDP given oracle access to a random permutation Π, Π^{-1}, and any $o(1)$-sequential VDE. A first attempt would be the system $\mathcal{C}$ which first samples $pp \leftarrow$ VDE.Setup(t, λ) for sufficiently large λ and then on input $(q, 0)$ returns $\iota(q) = \Pi(\textsf{VDE.Enc}(pp, \Pi(q)))$ and on input $(q, 1)$ returns $\iota^{-1}(q) =$

---

[9]Asymptotically t must be subexponential in λ. The reason for this is that the adversary needs to be able to run in time at least t, and if t is exponential in λ then the adversary might be able to break the computational security underlying σ-sequentiality. Of course, if σ-sequentiality is somehow achieved unconditionally then this constraint no longer applies.

$\Pi^{-1}(\mathsf{VDE.Dec}(pp, \Pi^{-1}(q)))$. We would further hope to argue that $\mathcal{C}$ is *indifferentiable* from $\mathcal{O}_{\mathsf{IDP}}^{(t)}$ in the sense of Maurer et. al. [38]. Unfortunately, due to the fact that Π used in the construction is necessarily available to the adversary as well, the adversary can choose any value (say 0) and query for $x = \Pi^{-1}(0)$, and then query C on input x. Since VDE is only necessarily sequential on *random* inputs the adversary may be able to find an input x in this way that causes C to output $\iota(x)$ in significantly fewer than t steps, which would enable it to distinguish C from $\mathcal{O}_{\mathsf{IDP}}^{(t)}$. Nonetheless, it is still possible that indifferentiability is achieved when the distinguisher is restricted to random queries (and for applications that only make random queries this may be good enough).

2.6 Depth Robust Graphs

An (n, α, β, d) *depth robust graph* (DRG) is a directed acyclic graph $G = (V, E)$ on $n = |V|$ nodes with regular in-degree d that has the property that any subgraph of at least αn nodes contains a path of length at least βn. We define the *depth* of a node v in a DRG, denoted $depth(v)$, as the longest path from some other node w in the graph to v. Depth robust graphs thus have the property that any sufficiently large subgraph contains some node of high depth in that subgraph. In fact, we can show that sufficiently large subgraphs contain many nodes of high depth.

Claim 1. *In any (n, α, β, d) DRG for constants $\alpha, \beta < 1$, every subgraph of size $(\alpha + \delta)n$ nodes contains at least $\lfloor \delta n \rfloor$ nodes of depth at least βn.*

Proof. Let $\alpha' = \alpha + \delta < 1$. Consider any subgraph G_1 of size $\alpha' n$. There exists a node $v_1 \in G_1$ of depth βn. Remove v_1 from G_1 to form the subgraph G_2. Continue this process, removing a node v_i of depth βn from the ith subgraph G_i, which can be found as long as $|G_i| = \alpha' n - i \geq \alpha n$. This process locates a sequence of distinct nodes $v_1 ..., v_t$ all of depth at least βn in G_1, where $t = \lfloor \delta n \rfloor$. $\qquad\square$

For example, if G is a $(n, 1/2, 1/4, d)$ DRG then at least 15% of the nodes in any subgraph on 50% of the nodes have depth at least $n/10$.

A directed graph has *B-bounded outdegree* if the outdegree of any node is bounded by some integer B. Not that any DAG with regular in-degree d has average out-degree d, however it may not have bounded outdegree. Intuitively, any DRGs with regular in-degree should have bounded out-degree, as otherwise one could delete many edges by removing only several nodes of high degree, reducing the depth of the remaining subgraph substantially.

Depth robust graph constructions A classical construction by Erdős, Graham, and Szemeredi [43] showed how to construct DRGs explicitly from extreme constant-degree bipartite expander graphs. Using their construction, one can obtain an $(n, \alpha, \beta, c \log n)$ DRG on n nodes for particular constants, e.g. $\alpha = 0.99$ and $\beta = 0.1$, and sufficiently large n. The constant factor c depends on the degree of the bipartite expander graphs used in the iterated construction. While explicit constructions of these graphs exist [41], they are complex and have either large constant degree or only achieve the requisite expansion properties for a significantly large number of nodes. Mahmoody, Moran, and Vadhan [36] use denser bipartite expander graphs to construct for any $\epsilon < 1$ a DRG family that is $(n, \alpha, \alpha - \epsilon, c \log^2 n)$ depth robust for all $\alpha < 1$. Again, instantiating this construction with explicit expanders will result in graphs that have an

impractically large degree. Alwen et. al. [6] recently improved the EGS construction to obtain DRGs for arbitrary α, β as well and $O(\log n)$ degree (an asymptotic improvement over MMV), however all of these constructions still require concretely high degree due to their use of bipartite expander graphs with extreme expansion properties.

A probabilistic DRG construction compromises on explicitness, and instead outputs a graph that is a DRG with overwhelming probability. This allows for much more efficient constructions. For example, one can immediately replace the bipartite expander graphs in the MMV/EGS constructions with *random bipartite graphs*, which are known to have strong expansion properties with high probability. Recently, Alwen et. al. [6] proposed and analyzed a highly efficient probabilistic DRG sampling algorithm that outputs concretely low-degree DRGs on n nodes with with failure probability negligible in n.

Another useful property of the ABH probabilistic DRG construction is that they are *locally navigatable*. That is, it outputs a graph equipped with an efficient parent function to derive the parents of any node in the graph using polylog time and space. A probabilistic instantiation of MMV with random bipartite graphs would also have this property. This is a critical property that we take advantage of in our construction as it allows our prover and verifier to interactively agree on a freshly sampled DRG with only $O(1)$ communication and $O(\text{polylog}(n))$ computation/space on the verifier. (In our application this help mitigate pre-computation attacks where the prover may find a specific seed input to the probabilistic sampling algorithm that outputs a graph that is not depth robust).

Definition 15. *A locally navigatable DRG sampling algorithm for an (n, α, β, d)-DRG is a pair of deterministic algorithms that share a common s-bit seed $\sigma \xleftarrow{\text{R}} \{0,1\}^s$ where $|s| = O(n \log n)$ that operate as follows:*

1. *DRG.Sample$(n, \sigma) \to G$ outputs a graph on the node set indexed by integers in $[n]$.*

2. *DRG.Parents$(n, \sigma, i) \to \mathcal{P}$ outputs a list $\mathcal{P} \subseteq [n]$ of the parents of the node at index $i \in [n]$ in the graph $G_\sigma \leftarrow DRG.Sample(n, \sigma)$.*

DRG.Sample(n, σ) runs in time $O(n \log n)$ and DRG.Parents(n, σ, i) runs in time $O(polylogn)$. Finally the graph G is an (n, α, β, d)-DRG with probability $1 - \text{negl}(n)$ over $\sigma \xleftarrow{\text{R}} \{0,1\}^s$.

In practice, the s-bit size of size $O(n \log n)$ can be derived pseudorandomly from a much smaller seed of size $O(\lambda)$ given a strong pseudorandom number generator (PRNG) with security parameter λ, or a random oracle. As n is presumed to be of feasible size, i.e. $O(\text{poly}(\lambda))$, we do not require PRNGs with exponential stretch and thus cryptographic PRGs built from any one-way function suffice.

In Section 5 we provide more details on our construction of locally navigatable (n, α, β, d)-DRGs as a generalization[10] of the Blocki et. al. construction to degree $d \geq 2$ and discuss its security both analytically and empirically.

2.7 Proofs of Space

A proof of space (PoS) [24] is an interactive proof between and prover and verifier that demonstrates the prover is storing an advice string of a minimum size. The framework for proofs of

---

[10]Our generalization is actually already implicit in their paper, as it is the "meta graph" they examine in their analysis.

space inherently requires a computational resource bound on the adversary. PoS constructions are built from puzzles that have time/space tradeoffs, and the PoS guarantees that an adversary who has not stored advice of a sufficient length will require computation exceeding this bound in order to pass the interactive proof. If the time/space tradeoff holds in the parallel computation model (i.e. passing the verifier's challenge with too few bits of advice is sequentially hard) then security can be enforced unconditionally by forcing the prover to respond to the verifier within a time bound.

We present here a slight variant on the PoS definition from [24] that additionally incorporates a data input D. This is a blend of the storage enforcing commitment concept and PoS. The PoS interactive protocol involves two phases and a setup procedure.

- **Setup** The setup runs on security parameters λ and outputs public parameters pp for the scheme. The public parameters are implicit inputs to the next two protocols.

- **Initialization** is an interactive protocol between a prover P and verifier V that run on shared input (id, N) and P is additionally given data D. P outputs Φ and S, where S is its storage advice and Φ is a compact $O(\text{polylog}(N))$ string given to the verifier.

- **Execution** is an interactive protocol between P and V where P runs on input S and V runs on input Φ. V sends challenges to P, obtains back a proof π, and outputs accept or reject.

Efficiency, Completeness, and Soundness   PoS protocols are required to have efficient verification, namely the verifier should run in time $O(\text{polylog}(N))$. The PoS protocol has perfect completeness if the verifier always outputs accept in interaction with an honest prover with probability 1 over the randomness of both parties. The PoS protocol is (s, t, μ)-sound if any adversarial prover P^* that runs in time complexity at most t and storage at most s during Execution, where $s = |S| \leq N$, passes verification with probability at most $\mu = \text{negl}(\lambda)$. We may also express that the protocol is *parallel* (s, t, μ)-sound if the time bound t is in parallel time (e.g. circuit depth, PRAM complexity, or parallel rounds of queries to an oracle). In either case there is no time bound on P's running time during Initialization, other than $O(\text{poly}(\lambda))$ when the scheme's security depends on computational assumptions about the setup parameters. We additionally say the PoS is *data committing* if the output Φ is a binding cryptographic commitment (not necessarily hiding) to the data input D.

Tight security and the space gap   The *space gap* in a PoS is the fraction $\epsilon = (N - s)/N$. This captures the difference between the (persistent) space utilization of the best possible attack and the honest prover's storage. A PoS with a tighter lower bound on the adversarial prover's space (persistent storage) will have a smaller space gap. As the best possible attack is probably not known, a PoS protocol with a larger space gap has a weaker security guarantee. Nearly all known PoS protocols to date have considerably large space gaps.

The original PoS of Dziembowski et. al. [24] leaves a space gap of at least $1 - \frac{1}{512}$. The construction of Ren and Devadas from stacked bipartite expander graphs made dramatic improvements, but still leaves space gap of at least $1/2$ (and much larger with practical parameters, e.g. their construction requires at least degree 40 graphs to achieve a space gap of less than $2/3$). This appears fundamental to the construction, which is also not secure against parallel attacks,

rather than a gap in the analysis. The recent proofs of space based on function tables[11] [3] has an enormous (even asymptotic) space gap of $1 - \frac{1}{64 \log N}$.

Most recently, Pietrzak constructed an asymptotically tight PoS relying on a strong property of the EGS depth robust graphs that was also recently proven [6]. This property (which is stronger than normal depth robustness) is that for any arbitrarily small ϵ a graph G_ϵ on N nodes with degree $O(\log N/\epsilon)$ can be constructed such that if any α fraction of nodes are removed the remaining nodes contain a path of length β such that $\alpha + \beta < 1 - \epsilon$. Using a graph with this property, Pietrzak's PoS then achieves a space gap of 4ϵ. However, the construction relies on bipartite graphs with extreme expansion called δ-local expanders[12] where $\delta < \epsilon/4$, and the final graph has degree at least $10c_\delta \log N$ where c_δ is the degree of the δ-local expander. Concretely, even the best known non-explicit randomized construction of a δ-expander (e.g. Chung's expander, which define edges by a random permutation [18]) would require degree greater than 40 for $\epsilon = 1/2$ and degree greater than 276 for $\epsilon = 1/8$. To achieve just an $1/2$ space gap would thus require a graph with degree $d = 2,760 \log N$. Even for 10-bit security the proof size would be at least 26MB![13]

2.8 Graph pebbling games

Pebbling games are the main analytical tool used in graph-based proofs of space and memory hard functions.

Black pebbling game   The black pebbling game is a single-player game on a DAG $\mathcal{G} = (V, E)$. At the start of the game the player chooses a starting configuration of $P_0 \subseteq V$ of vertices that contain black pebbles. The game then proceeds in rounds where in each round the player may place a black pebble on a vertex only if all of its parent vertices currently contain pebbles placed in some prior round. In this case we say that the vertex is *available*. Placing a pebble constitutes a *move*, whereas placing pebbles on all simultaneously available vertices consumes a *round*. The adversary may also remove any black pebble at any point. The game stops once the adversary has placed pebbles on all vertices in some target/challenge set $V_C \subseteq V$.

Pebbling complexity   There are different pebbling complexity measures taking into account how many rounds, steps, and/or pebbles are needed to finish the game from any given starting configuration. We say that the pebbling game on graph G with vertex set V and target set $V_C \subseteq V$ is (s,t)-*hard* if no player can pebble the set V_C in t moves (or fewer) starting from s initial pebbles, and is (s,t)-*parallel-hard* if no player can complete the pebbling in t rounds (or fewer) starting from an initial configuration of at most s pebbles. These are the two measures we are most interested in. If the hardness holds for any subset containing an α fraction of V_C then we write that the pebbling game on (G, V_C) is (s,t,α)-(parallel)-hard. From a fixed starting configuration P_0 the number of parallel rounds required to pebble all nodes in the set V_C is equal to the maximum number of parallel rounds requires to pebble any individual $v \in V_C$.

---

[11]These are the proofs of space that are being deployed by Chia Network, `https://chia.net/`.

[12]Bipartite δ-local expanders require the property that there is an edge between any δ fraction of sinks and δ fraction of sources.

[13]The proof size is at least $O(\lambda d)$, so even for $\lambda = 10$ (i.e. 10 bit security) and $N = 2^{30}$ (i.e. a 32GB PoS because each node is a 32 bytes value) the proof size would be at least 26MB. Furthermore, as ϵ decreases $\lambda = O(1/\epsilon)$ to maintain the same security level. To achieve a space gap of $1/10$ with 10-bit security would require $d = 19,310 \log N$ and $\lambda = 60$ for a proof size of at least 1GB.

Random pebbling games are also considered, where a challenge node is sampled randomly from V_C after the player commits to the initial configuration P_0 of s vertices, and the hardness measure includes the adversary's probability of success. The random pebbling game is (s, t, ϵ)-(parallel)-hard if from any s fixed initial pebbles the probability that a uniformly sampled challenge node can be pebbled in t or fewer moves (resp. t or fewer rounds) is less than ϵ. There is also a general correspondence between the parallel hardness of random and deterministic pebbling games on a given graph.

Claim 2. *The random pebbling game on a DAG G on n nodes with target set V_C is (s, t, α)-parallel-hard if and only if the deterministic pebbling game on G with target set V_C is (s, t, α)-parallel-hard.*

Proof. Fix any P_0 of size s. If the random pebbling game on G with target set V_C is (s, t, α)-parallel-hard then less than an α fraction of the nodes in V_C can be pebbled individually in t rounds starting from P_0. Therefore, every subset U in V_C of size $\alpha|V_C|$ contains at least one node that cannot be pebbled individually in t rounds, hence the (deterministic) pebbling game is (s, t, α)-parallel-hard. Conversely, if G is (s, t, α)-parallel-hard then less than an α fraction of nodes in V_C can be pebbled individually in r rounds. Otherwise, these nodes form a subset U of size $\alpha|V_C|$ and they can all be simultaneously pebbled in parallel in t rounds. This implies that the probability a randomly sampled node from V_C can be pebbled in t rounds is less than α. $\qquad\square$

Claim 3. *A random pebbling game with a single challenge is (s, t, α)-parallel-hard if and only if the the random pebbling with κ challenges is (s, t, α^k)-parallel-hard.*

Proof. If the random pebbling game is (s, t, α) hard then by Claim 2 the deterministic pebbling game is (s, r, α) hard, hence there are at most an α fraction of the nodes in V_C that can be pebbled in r rounds from s initial pebbles. The probability that κ independent random challenges are all nodes from this α fraction is at most α^κ. Conversely, if the random pebbling game is not (s, t, α) hard then the adversary can pebble all the κ challenges simultaneously in parallel time t succeeding on each challenge individually with probability greater than α, hence succeeding on all the challenges with probability greater that α^κ. $\qquad\square$

From pebbling to graph labeling A labeling game on a degree d DAG $\mathcal{G}$ is analogous to the pebbling game except that the pebbling rules are "enforced" using a cryptographic hash function $H : \{0, 1\}^{dm} \to \{0, 1\}^m$, often modeled as a random oracle. The vertices of the graph are indexed in $[n]$ and each ith vertex associated with the label c_i where $c_i = H(i)$ if i is a source vertex, or otherwise $c_i = H(i||\ell_{\mathsf{parents}}(i))$ where $\ell_{\mathsf{parents}}(i) = \{c_{v_1}, ..., c_{v_d}\}$ if $v_1, ..., v_d$ are the parents of the ith vertex, i.e. the vertices with a directed edge to vertex i. The game ends when the player has computed all the labels on a target/challenge set of vertices V_C. A "fresh" labeling of $\mathcal{G}$ could be derived by choosing a salt id for the hash function so that $H_{id}(x) = H(id||x)$, and the labeling may be associated with the identifier id. The complexity of the labeling game (on a fresh identifier id) is measured analogously to the pebbling game, but in terms of queries to the hash function instead of pebbles. This includes the number of labels initially stored, the total number of queries, and the total rounds of sequential queries, etc. The labeling game is $(s, r, q, \epsilon, \delta)$-labeling-hard if no algorithm that stores initial advice of size s and after receiving a uniform random challenge node $v \in [n]$ makes a total of q queries to H in r sequential rounds

can output the correct label on v with probability greater than ϵ over the challenge v and δ over the random oracle H.

Random oracle query complexity While the pebbling complexity of the underlying graph $\mathcal{G}$ provides strong intuition about the query complexity of the labeling game, proving this equivalence is highly non-trivial. The "ex post facto" technique of [20] allows one to prove this when the initial configuration of pebbles is empty and the target set V_C is chosen deterministically. Progress towards a general correspondence was made in [29, 45]. In particular, Pietrzak [45] showed an equivalence between the parallel hardness of the randomized pebbling game and the parallel hardness of the random oracle labeling game for arbitrary initial configurations S_0. Demonstrating (or refuting) an equivalence for the non-parallel hardness of the two games is still open.

Theorem 2 (Pietrzak [45]). *If the random pebbling game on a DAG G with n nodes and in-degree d is (s, r, ϵ)-parallel-hard then the labeling game on G with a random oracle $H : \{0, 1\}^{md} \to \{0, 1\}^m$ is $(s', r, \epsilon, \delta, q)$-labeling-hard with $s' = s(m - 2(\log n + \log q)) - \log(1/\delta)$.*

Generic PoS from graph labeling game Most PoS constructions are based on the graph labeling game [24, 45, 46]. Let $\mathcal{G}(\cdot)$ be a family of d-in-regular DAGs such that $G_n \leftarrow \mathcal{G}(n)$ is a d-in-regular DAG on $N > n$ nodes and $V_C(n)$ is a subset of n nodes from G_n. Let $H : \{0, 1\}^{dm} \to \{0, 1\}^m$ be a collision-resistant hash function (or random oracle). Let $\mathsf{Chal}(n)$ denote a distribution over $[N]^\lambda$. For each $n \in \mathbb{N}$, the generic PoS based on the labeling game with G_n and target set $V_C(n)$ is as follows:

1. Initialization: The prover plays the labeling game on G_n using a hash function H salted with the proof identifier id, i.e. $H_{id} = H(id||\cdot)$. While computing the labels on all nodes of $\mathcal{G}$, the prover adds these to a vector commitment (e.g. Merkle tree) denoted com. Then it derives λ non-interactive challenges by sampling a vector $\vec{c} \xleftarrow{\text{R}} \mathsf{Chal}(n)$ using as a seed $H_{id}(com)$. For each of the challenges $c_1, ..., c_\lambda$, the prover opens the label on the c_ith node of G_n, which was committed in com, as well as the labels of all its parent nodes. The labels are added to a list L with corresponding opening proofs in a list Λ and the prover outputs the non-interactive proof $\Phi = (com, L, \Lambda)$. A verifier (at any point in time) accepts Φ if it successfully verifies the openings Λ with respect to com and also that for each challenge $v \in [N]$ with corresponding ℓ_v and purported parent labels $\ell_1^{(v)}, ..., \ell_d^{(v)}$ listed in L that $\ell_v = H_{id}(v||\ell_1^{(v)}|| \cdots ||\ell_d^{(v)})$. Finally, the prover stores as S only the n labels in V_C.

2. Execution: The verifier selects κ challenge nodes $v_1, ..., v_\kappa$ uniformly at random from V_C. The online prover uses its input S to respond with the label on v and an opening of com at the appropriate index.

Roughly, the generic construction is a PoS when the pebbling game on G and V_C is "hard" in some pebbling complexity sense. Our security analysis will focus on parallel pebbling complexity as this is a stronger security notion, particularly considering its general correspondence to the parallel hardness of the labeling game in the random oracle model.

Red-black pebbling game The soundness of the generic labeling PoS is a bit more nuanced than just the parallel hardness of the black pebbling game on $\mathcal{G}$ with V_C. It is best abstracted

through the red-black pebbling game, where red pebbles correspond to incorrect labels that the adversary computes and includes in its commitment during Initialization (i.e. where the adversary cheats). We may assume without loss of generality that whenever the adversary cheats it generates some label that does not require any space to store, essentially placing a "free" pebble on this node. Naturally, by committing to the labels the adversary commits to all these red pebbles during Initialization. The adversary's choice of red pebble placements is also constrained by the λ non-interactive challenges, which may catch these red pebbles and reveal them to the verifier. The formal description of the red-black pebbling security game for a graph labeling PoS construction with $\mathcal{G}(n)$, $V_C(n)$, and $\mathsf{Chal}(n)$ is as follows.

Red-Black-Pebbles$^{\mathcal{A}}(\mathcal{G}, V_C, \mathsf{Chal}, t)$:

1. $\mathcal{A}$ outputs a set $R \subseteq [N]$ (of red pebble indices) and $S \subseteq [N]$ (of black pebble indices).

2. The challenger samples $c_1, ..., c_\lambda \xleftarrow{\mathrm{R}} \mathsf{Chal}(n)$. If $c_i \in R$ for some i then $\mathcal{A}$ immediately loses. The challenger additionally samples $v_1, ..., v_\kappa$ uniformly at random from indices in $V_C(n)$ and sends these to $\mathcal{A}$.

3. $\mathcal{A}$ plays the random (black) pebbling game on $\mathcal{G}(n)$ with the challenges $v_1, ..., v_\kappa$ and initial pebble configuration $P_0 = R \cup S$. It runs for t parallel rounds and outputs its final pebble configuration P_t. $\mathcal{A}$ wins if P_t contains pebbles on all of $v_1, ..., v_\kappa$.

Although we have been loosely referencing PoS (S, T)-soundness, we now formally define this for the special case of any graph labeling PoS in terms of complexity of Red-Black-Pebbles$^{\mathcal{A}}(\mathcal{G}, V_C, t)$. Let $c : \mathbb{N} \to N$ denote a cost function $c : \mathbb{N} \to \mathbb{N}$ representing the parallel time cost (e.g. in sequential steps on a PRAM machine) of computing a label on a node of $\mathcal{G}(n)$ for each $n \in \mathbb{N}$.

Definition 16. *A graph labeling PoS with $\mathcal{G}(n), V_C(n), \mathsf{Chal}(n)$ and cost function $c(n)$ is parallel $(s, c(n) \cdot t, \mu)$-sound if and only if the probability that any $\mathcal{A}$ wins Red-Black-Pebbles$^{\mathcal{A}}(\mathcal{G}, V_C, \mathsf{Chal}, t)$ is bounded by μ where $|S| = s$.*

Composition For the security of several graph labeling games to compose the independent instances of the games must involve either independent labelings or independent hash function. This is the function of id in the generic PoS construction from a graph labeling, which specifies a fresh hash function H_{id}. In the random oracle model this is represented as a freshly sampled random function. When this is the case the composition of k independent games is equivalent to a single game on a large graph containing k distinctly labeled copies of the graph, where the new distribution of Chal is a k fold direct product.

Definition 17. *An $(s, c(n) \cdot t, \mu)$-sound graph labeling PoS with $\mathcal{G}(n), V_C(n), \mathsf{Chal}(n)$ and cost function $c(n)$ is secure under k-fold composition if and only if the graph labeling PoS with $\mathcal{G}^*(n), V_C^*(n), \mathsf{Chal}^*(n)$ is $(sk, c(n) \cdot t, \mu)$-sound, where $\mathcal{G}^*(n)$ consists of k copies of $\mathcal{G}(n)$, $V_C^*(n)$ is the union of the images of $V_C(n)$ in each copy, and $\mathsf{Chal}^*(n)$ is the direct product distribution $\mathsf{Chal}_1(n) \times \cdots \times \mathsf{Chal}_k(n)$ where each $\mathsf{Chal}_i(n)$ is the distribution $\mathsf{Chal}(n)$ projected onto the ith copy of the graph.*

Uniform challenges   In the special case that Chal is the uniform distribution over $[N]^\lambda$ then the hardness of Red-Black-Pebbles$^\mathcal{A}(\mathcal{G}, V_C, \mathsf{Chal}, t)$ is simply implied by the hardness of the random black pebbling game.

Claim 4. *If the random black pebbling game on $\mathcal{G}(n)$ is $((\gamma + \delta)n, t, \mu)$-parallel-hard and Chal is the uniform distribution over $[N]^\lambda$ then the graph labeling PoS with $\mathcal{G}(n)$, $V_C(n)$, and $c(n)$ is $(\gamma n, c(n) \cdot t, max((1 - \delta)^\lambda, \mu^\kappa))$-sound.*

Proof. The λ challenges ensure that that $\mathcal{A}$ wins the game Red-Black-Pebbles$^\mathcal{A}(\mathcal{G}, V_C, \mathsf{Chal}, t)$ with probability at most $(1 - \delta)^\lambda$ if the number of red pebbles (i.e. $|R|$) exceeds δN. If $|R| \leq \delta N$ and $|S| \leq \gamma N$, then the adversary plays the random black pebbling game with κ challenges and an initial configuration of at most $(\gamma + \delta)N$ pebbles. If the random black pebbling game (with a single challenge) is $((\gamma + \delta)N, t, \mu)$-parallel-hard then $\mathcal{A}$ succeeds in playing this game with κ simultaneous challenges (Step 3) with probability at most μ^κ (by Claim 3). Therefore, if the random black pebbling game is hard then the adversary's overall success probability is bounded by the maximum of its success probability in the two cases regarding the size of R, i.e. $max(\mu^\kappa, (1 - \delta)^\lambda)$. $\qquad\square$

In this case we can also make a simple claim about the composition of a graph labeling PoS where Chal is the uniform distribution. It follows from a general claim pertaining a graph $\mathcal{G}(n)$ that is the composition of k subgraphs $\mathcal{G}_1(n), ..., \mathcal{G}_k(n)$ which all have the same random black pebbling complexity and Chal samples λ challenges from the uniform distribution on each.

Claim 5. *If $\mathcal{G}(n)$ contains subgraphs $\mathcal{G}_1(n), ..., \mathcal{G}_k(n)$ and for each i the random black pebbling game on $\mathcal{G}_i(n)$ is $((\gamma + \delta)n, t, \mu)$-parallel-hard and $\mathsf{Chal} = \mathsf{Chal}_1 \times \cdots \times \mathsf{Chal}_k$ where Chal_i is the uniform distribution over the indices of the nodes contained in $\mathcal{G}_i$ then the graph labeling PoS with $\mathcal{G}(n)$, $V_C(n)$, and $c(n)$ is $(k\gamma n, c(n) \cdot t, max((1 - \delta)^\lambda, \mu^\kappa))$-sound.*

Proof. The λ challenges sampled from each Chal_i ensure that that $\mathcal{A}$ wins the game Red-Black-Pebbles$^\mathcal{A}(\mathcal{G}, V_C, \mathsf{Chal}, t)$ with probability at most $(1 - \delta)^\lambda$ if the number of red pebbles exceeds δN in *any* subgraph $\mathcal{G}_i$. Suppose there are $\gamma_i N$ black pebbles initially on each $\mathcal{G}_i$ and at most δN red pebbles (total of $\sum_i \gamma_i$ initial black pebbles overall. The adversary plays the random black pebbling game with κ challenges on each $\mathcal{G}_i$ with an initial configuration of at most $(\gamma_i + \delta)N$ pebbles. Since the random black pebbling game on each $\mathcal{G}_i$ is $((\gamma + \delta)N, t, \mu)$-parallel-hard then $\mathcal{A}$ succeeds in playing this game with κ simultaneous challenges (Step 3) with probability at most μ^κ if $\gamma_i < \gamma$. The probability $\mathcal{A}$ succeeds in all the k games is bounded by the minimum of its success probabilities in each, hence its overall success probability is at most μ^κ if $\gamma_i < \gamma$ for *any* i, i.e. if $\sum_i \gamma_i < k\gamma$. Therefore, if the total number of initial pebbles is less than $k\gamma$ then the adversary's overall success probability is bounded by $max(\mu^\kappa, (1 - \delta)^\lambda)$. $\qquad\square$

Pebbling DRGs   We now state several facts regarding the pebbling complexity of a depth robust graph.

Claim 6. *If G is a (n, α, β)-depth-robust DAG then pebbling any αn initially unpebbled nodes requires at least βn rounds.*

Proof. A subgraph of unpebbled αn nodes contains a path P of length βn. To pebble the entire set of unpebbled αn nodes requires sequentially pebbling all the βn nodes along the path P. $\qquad\square$

Claim 7. *For any $\alpha < \mu < 1$, the random black pebbling game on an (n, α, β)-depth-robust DAG G is $((\mu - \alpha)n, \beta n - 1, \mu)$-parallel-hard.*

Proof. Consider any $\mu > \alpha$. If $(\mu - \alpha)n$ black pebbles are placed anywhere on the graph then at least $(1 + \alpha - \mu)n$ nodes are missing pebbles. Any subset of μn nodes of the graph therefore include at least αn nodes that are missing pebbles. By Claim 6 these subsets require at least βn rounds to pebble. This means that the pebbling game on G is $((\mu - \alpha)n, \beta n - 1, \mu)$-parallel-hard. The random pebbling game on G is equivalently hard by Claim 2. $\square$

PoS from DRGs The claims established thus far imply PoS soundness of the generic graph labeling PoS on a graph G that is (n, α, β, d) depth robust, with challenge set $V_C = [n]$, and where Chal is the uniform distribution over $[n]^\lambda$. By Claim 4 and Claim 7, this PoS is $((\mu - \alpha - \delta)n, c(n) \cdot (\beta n - 1), max((1 - \delta)^\lambda, \mu^\kappa))$-sound for any $\mu > \alpha + \delta$. In particular, setting $\kappa = \lambda$ and $\mu = 1 - \delta$, the PoS is $((1 - \alpha - 2\delta)n, c(n) \cdot (\beta n - 1), (1 - \delta)^\lambda)$-sound. As δ can be made arbitrarily small for an increase in λ, the space gap of this PoS is approximately α. Theoretically, there are DRG constructions that achieve arbitrarily small α which would seem to imply tight space-hardness. However, this space-hardness is not really tight as decreasing α decreases the space gap but also reduces the time bound $t = \beta n$).

Lemma 2. *The graph labeling PoS on a family $G(n)$ of (n, α, β) depth robust graphs with labeling cost function $c(n)$ is an (s, t, μ)-parallel-sound PoS where for any $\delta < 1$:*

$$s = \gamma n \qquad t = c(n)(\beta n - 1) \qquad \mu = max((1 - \delta)^\lambda, (\alpha + \delta + \gamma)^\kappa)$$

The composition of k independent graph labeling PoS protocols on this family of depth robust graphs is a $(k \cdot s, t, \mu)$-parallel-sound PoS.

Proof. As described above, this follows from Claim 4 and Claim 7, setting $\gamma = \mu - \alpha - \delta$. Composition follows from Claim 5 because this is equivalent to a graph labeling PoS on $G^*(n)$ that is the concatenation of k copies of $G(n)$ and $\mathsf{Chal} = \mathsf{Chal}_1 \times \cdots \mathsf{Chal}_k$ where each Chal_i is the uniform distribution over $[i \cdot n, (i + 1)n]^\lambda$. $\square$

3   Proof-of-Replication

We begin with defining the syntax for a proof-of-replication (PoRep). A PoRep operates on arbitrary data $D \in \{0, 1\}^*$ of up to $O(\text{poly}(\lambda))$ size for a given security parameter λ. All algorithms are assumed to operate in the RAM model of computation (in particular reading a bit of an input is assumed to be an $O(1)$ operation). Parallel algorithms operate in the PRAM model.

1. $\mathsf{PoRep.Setup}(\lambda, T) \to pp$ is a one-time setup that takes in a security parameter λ, time parameter T, and outputs public parameters pp. T determines the challenge-response period.[14]

---

[14]An alternative formulation may allow for a variable challenge-response period T that might depend on the size of the data input. Our formulation does not allow this, implying that the online prover $\mathsf{PoRep.Prove}$ must run in parallel time *less than* T (including transmission time of the its proof to the verifier) independent of the data size (up to $O(\text{poly}(\lambda))$ in length). Achieving a fixed polling period T for all input data of any reasonable length (i.e.poly(λ)) is important in practice.

2. PoRep.Preproc$(sk, D) \to \tilde{D}, \tau_D$ is a preprocessing algorithm that may take a secret key sk along with the data input D and outputs preprocessed data $\tilde{D}$ along with its data tag τ_D, which at least includes the size $N = |D|$ of the data. The preprocessor operates in *keyless mode* when $sk = \perp$.

3. PoRep.Replicate$(id, \tau_D, \tilde{D}) \to R$, aux takes a replica identifier id and the preprocessed data $\tilde{D}$ along with its tag τ_D. It outputs a replica R and (compact) auxilliary information aux which will be an input for the Prove and Verify procedures. (For example, aux could contain a proof about the replication output or a commitment).

4. PoRep.Extract$(pp, id, \tau_D, R) \to \tilde{D}$ on input replica R and identifier id and data tag τ_D outputs the data $\tilde{D}$ (and verifies its consistency with τ_D).

5. PoRep.Prove$(R, \text{aux}, id, r) \to \pi$ on input replica R, auxilliary information aux, replica identifier id, and challenge r, outputs a proof π_{id}.

6. PoRep.Poll(aux) $\to r, s$: This takes as input the auxiliary replica information aux and outputs a public challenge r along with any private randomness s used to produce this challenge. For public coin constructions $s = \perp$.

7. PoRep.Verify$(id, \tau_D, r, s, \text{aux}, \pi) \to \{0, 1\}$ on input replica identifier id, data tag τ_D, public challenge r, private randomness s (used to produce r), auxilliary replication information aux, and proof π it outputs a decision to accept (1) or reject (0) the proof.

PoRep interactive protocol These algorithms are used in an interactive protocol as illustrated in Figure 1. A PoRep protocol between a verifier $\mathcal{V}$, a prover $\mathcal{P}$, and data input D with identifier id on setup parameters $pp \leftarrow$ PoRep.Setup(λ, T) is denoted as the system PoRep$(\mathcal{P}(pp, id, D) \leftrightarrow \mathcal{V}(pp, id))$. The setup (whether a deterministic, trusted, or transparent public setup, such as one involving a public random source) is run externally and pp is given as an input to all parties. For each file D, a preprocessor (a special party or the prover when operating in keyless mode, but not the verifier) runs $(\tilde{D}, \tau_D) \leftarrow$ PoRep.Preproc(sk, D). The outputs $\tilde{D}, \tau_D$ are inputs to the prover and τ_D to the verifier.

Transparency, public coin, and public verifiability A PoRep scheme may involve a trusted one-time setup, in which case PoRep.Setup is run by a trusted party[15] and the output pp is published for all parties to see. A *transparent* PoRep scheme is one in which the setup does not involve any private information. This trusted setup is an independent, one-time procedure, and the trusted party that runs the setup should have no further involvement in the interactive protocol. The data preprocessor on the other hand may use a secret-key, but it is not trusted. In particular, the verifier is oblivious to whether the preprocessor colludes with the server or not. As we will discuss next the secret-key preprocessor only has implications for data retrievability, but not for the security of the publicly verifiable data replication (or proof of space). This is an important distinction from the notion of proof of data replication proposed in the system Mirror. A PoRep scheme may be *private coin* if the private input r to PoRep.Poll is non-empty, and it is *public coin* if $r = s$. A public coin PoRep has the additional desirable property that the

---

[15]As usual, the trusted party can also be replaced with a committee that runs a multi-party computation (MPC) protocol to generate the public parameters

Prover	**Verifier**

Replication Phase

1 : $R, \mathsf{aux} \xleftarrow{\text{R}} \mathsf{PoRep.Replicate}(id, \tau_D, \tilde{D})$

$$id, \mathsf{aux} \longrightarrow$$

Challenge-Response Phase

2 : $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r, s \xleftarrow{\text{R}} \mathsf{PoRep.Poll}(\mathsf{aux})$

$$\longleftarrow r$$

3 : $\pi \xleftarrow{\text{R}} \mathsf{PoRep.Prove}(R, \mathsf{aux}, id, r)$

$$id, \pi \longrightarrow$$

4 : $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad b \xleftarrow{\text{R}} \mathsf{PoRep.Verify}(id, \tau_D, r, s, \mathsf{aux}, \pi)$
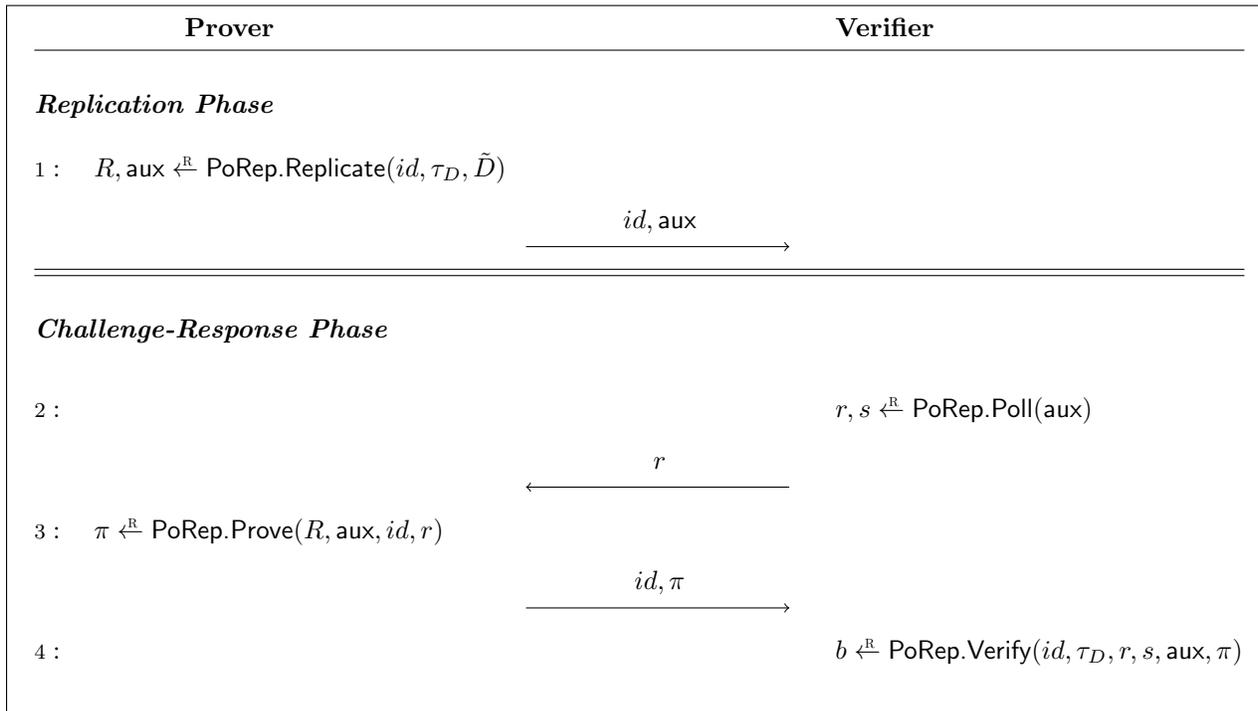
Figure 3.1: The diagram illustrates the interaction between a prover and verifier in a PoRep protocol. The setup and data preprocessing is run externally generating $pp \leftarrow \mathsf{PoRep.Setup}(\lambda, T)$ and $\tilde{D}, \tau_D \leftarrow \mathsf{PoRep.Preproc}(sk, D)$. The challenge-response protocol is timed, and the verifier rejects any response that is received more than T time steps after sending the challenge. This is formally captured by requiring $\mathsf{PoRep.Prove}$ to run in parallel time at most T. The propogation delay on the communication channel between Prover and Verifier is assumed to be nominal in comparison to T.

challenge output by $\mathsf{PoRep.Poll}$ could be replaced by a challenge derived from an unpredictable public random beacon[16].

Data preprocessing and data retrievability   When the data input is preprocessed using a secret-key the resulting PoRep is a public-coin PoR (among satisfying other properties regarding replication and proof of space that we will discuss further in the following section on the PoRep security model). In this scenario we can imagine the preprocessor is a single client who wants to store (and replicate) data on a server, and generates the data tag τ_D to outsource this verification work. When the preprocessor runs in keyless mode the resulting PoRep is a PoRC (Section 2.4) with respect to the output τ_D, which includes a commitment to the data D. Any (stateful) verifier that is at one point given the opening of the commitment can thereafter use the tag to verify PoReps as standard PoRs. This is particularly useful for a setting in which multiple clients pool their files together and want to receive a single PoRep for the entire dataset, but they do not mutually trust one another to share a private-key. It is also appropriate for a dynamic

---

[16]*Proofs-of-space-time* [2] is a proposed construction that non-interactively chains several PoReps by using the output of the last PoRep as a seed to the next, and would only work for an underlying PoRep that is public coin.
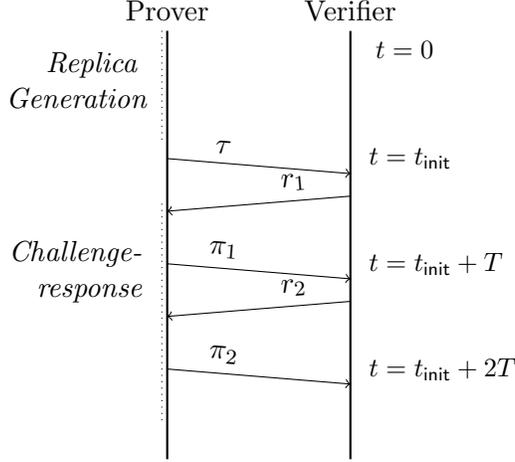
Figure 3.2: Space-time diagram of the PoRep protocol. Following a phase of length t_{init} during which the prover generates a new replica, the verifier repeatedly challenges the prover to produce a PoRep within a challenge time period length T in order to verify that the prover is still storing the unique replica of the original data. For this proof system to be sound it is necessary that $t_{\mathsf{init}} >> T$.

setting where new clients are made aware of the data stored on the server and wish to verify retrievability without trusting the original client's private-key preprocessing.

Treating data preprocessing in this way as a separate "layer of the stack" allows for a wider variety of constructions appropriate in different settings without impinging on the efficiency of the underlying PoRep protocol itself. In fact, a PoRep protocol that (without preprocessing) satisfies $(\delta, \mathcal{C})$-PoRC (e.g. for $\mathcal{C}$ partitioning the data into constant size blocks) can immediately be compiled into either a public-key PoR or a PoRC using any number of previous techniques. We highlight the two scenarios:

1. A client has a data file D and uses a randomly generated secret key sk to produce $\tilde{D}, \tau_D \leftarrow$ PoRep.Preproc(sk, D) which applies a rate-δ adversarial erasure code [15] to D (i.e. one that is resilient to arbitrary adversarial deletions of up to δ fraction of the blocks in $\tilde{D}$). The tag τ_D is a commitment to $\tilde{D}$. The adversarial erasure code may involve a secret transformation using sk and requires sk as well to recover from these errors. The private-key setting enables using more efficient erasure codes that only tolerate random deletions because they can be combined with a secret pseudorandom permutation of the data so that an adversary will only succeed in targeted deletions with up to negligible probability [15,48]. The client then uses τ_D for verification in the PoRep protocol, which we assume is already a $(\delta, \mathcal{C})$-PoRC with respect to τ_D. Any verifier $\mathcal{V}$ can publicly verify that a δ fraction of the committed $\tilde{D}$ is retrievable (and indeed there is a public extraction algorithm for a δ fraction of blocks in $\tilde{D}$), which is sufficient for the private client to recover D using sk.

2. A keyless preprocessor transforms the data file D into a single code word of a deterministic rate-δ optimal erasure code (e.g., Reed-Solomon) with symbol size equal to the block size in $\mathcal{C}$. It outputs τ_D as a deterministic commitment to $\tilde{D}$, which is also a deterministic commitment to D. Due to the resilience of the erasure code (D can be recovered from any δ fraction of the symbols in $\tilde{D}$), the PoRep $(\delta, \mathcal{C})$-PoRC of $\tilde{D}$ becomes a PoRC of D.

Efficiency All algorithms must run in time $O(\text{poly}(\lambda))$ and with up to at most $O(\text{poly}(\lambda))$ parallelism. We don't directly impose further efficiency requirements in the PoRep definition, but implicitly a PoRep scheme cannot be simultaneously correct and secure unless PoRep.Poll and PoRep.Prove have a lower parallel time complexity than PoRep.Replicate on $O(\text{poly}(\lambda))$ processors. Otherwise, for correctness to hold the challenge-response period will be long enough for an adversarial prover to rerun the replica generation from scratch. Furthermore, for practical purposes it is highly desirable for PoRep.Prove and PoRep.Verify to run in constant or $O(\text{polylog}(|D|))$ time, where $|D|$ is the length of the data file. Among other benefits, this enables batched proofs and verification of multiple replicas of the same (or distinct) files, which scales sublinearly in the number of replicas/files. Replica generation PoRep.Replicate runs in time greater than $max(T, |D|)$, however it is desirable for this runtime to be sublinear in $T \cdot |D|$ as the size of the file increases.

Correctness In a correct PoRep construction, a prover who runs PoRep.Replicate and PoRep.Prove honestly must pass verification with probability 1. In order to pass verification the algorithm PoRep.Prove must run in parallel time at most T as otherwise the verifier will not receive the output within the challenge-response period. In addition, PoRep.Extract must correctly extract the original data from a replica generated with PoRep.Replicate.

Definition 18 (PoRep correctness). *A PoRep construction is correct if for any security parameter λ, any id and data input D, there exists T_λ such that for all $T \geq T_\lambda$, $pp \leftarrow$ PoRep.Setup(λ, T), and $(R, \tau) \leftarrow$ PoRep.Replicate(pp, id, D) it holds that:*

$$Pr\left[\text{PoRep.Verify}(pp, id, r, s, \tau, \pi) = Accept \ \middle| \ \begin{array}{l} (r, s) \leftarrow \text{PoRep.Poll}(pp, \tau) \\ \pi \leftarrow \text{PoRep.Prove}(pp, R, id, r) \end{array}\right] = 1$$

and PoRep.Prove(pp, R, id, r) *runs in parallel time at most* T_λ.[17]
Additionally:

$$Pr\left[\tilde{D} = D \ \middle| \ \begin{array}{l} R \leftarrow \text{PoRep.Replicate}(pp, id, D) \\ \tilde{D} \leftarrow \text{PoRep.Extract}(pp, id, R) \end{array}\right] = 1$$

3.1 Replication Security Model

An ideal security goal for PoRep protocols, beyond satisfying the standard definition of a PoR, would be to guarantee the following properties, described informally:

1. Any prover who simultaneously passes verification in k distinct PoRep protocols (under k distinct identities) where the input to PoRep.Replicate is a file D_i in the ith protocol must be storing k independent replicas, one for each D_i, even if several of the files are identical.

2. A prover who passes verification in a PoRep protocol where the input to PoRep.Replicate is a file $\tilde{D} = D \circ \cdots \circ D$ (consisting of k copies of the same file D) must be storing k independent physical copies of D.

Adverary	**Challenger**
1: $\mathcal{A}_{\text{pre}}$ chooses $\vec{D} = D_1, ..., D_k$; $\vec{id} = id_1, ..., id_k$; and computes data tags $\vec{\tau} = \tau_{D_1}, ..., \tau_{D_k}$	
2: $\sigma, \vec{\text{aux}} \leftarrow \mathcal{A}_{\text{pre}}(pp, z, \vec{id}, \vec{\tau}, \vec{D})$	

$$\xrightarrow{\quad z, \vec{id}, \vec{\text{aux}} = \text{aux}_1, ..., \text{aux}_k \quad}$$

| | 3: $\forall i\ r_i \xleftarrow{\text{R}} \text{PoRep.Poll}(\text{aux}_i)$ |

$$\xleftarrow{\quad \vec{r} = r_1, ..., r_k \quad}$$

| 4: $\hat{z}, \vec{\pi} \leftarrow \mathcal{A}_{\text{onl}}(pp, \sigma, \vec{\text{aux}}, \vec{id}, \vec{r})$ | |

$$\xrightarrow{\quad \vec{\pi} = \pi_1, ..., \pi_k \quad}$$

| | 5: $b_i \leftarrow \text{PoRep.Verify}(id_i, \tau_{D_i}, r_i, \text{aux}_i, \pi_i)$ |
| Output $\hat{z}$ | Output $\hat{b} = \bigwedge_i b_i$ |
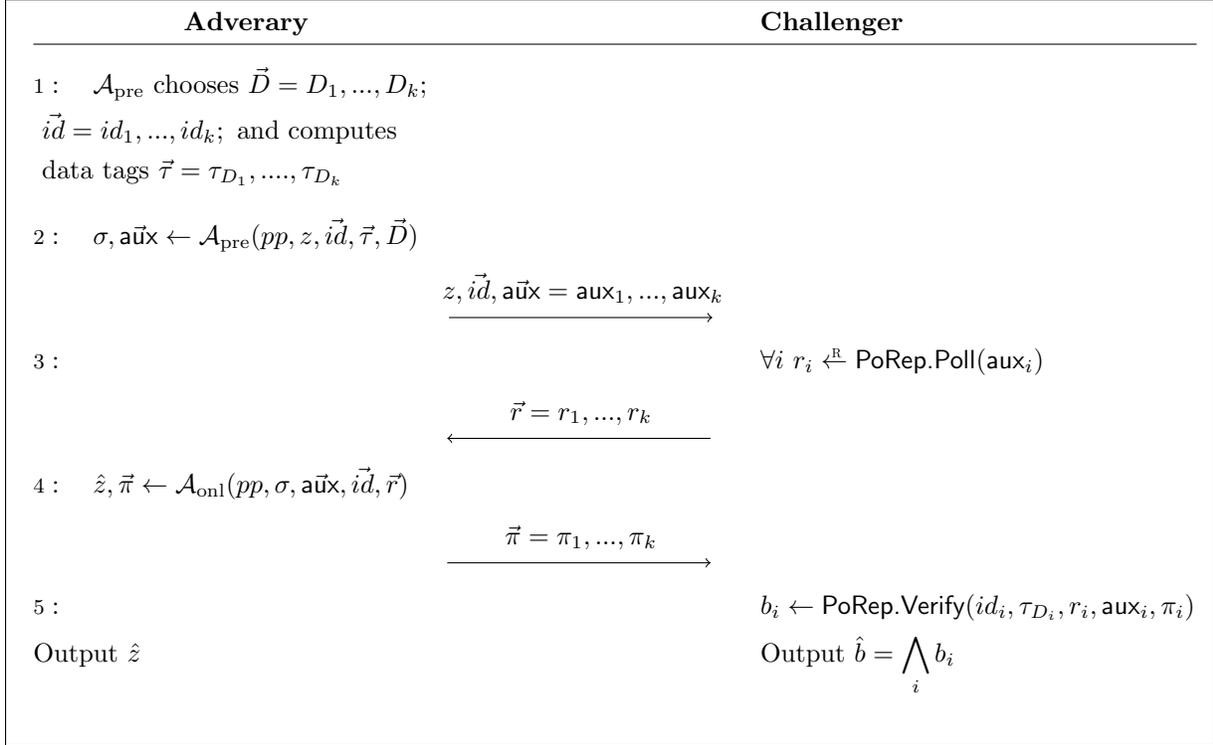
Figure 3.3: The diagram illustrates the PoRep security game between an adversary and a challenger. Several inputs come from the environment, which runs the setup $pp \leftarrow \text{PoRep.Setup}(\lambda, T)$ and chooses an auxiliary input z for the adversary. The adversary is split into two components: $\mathcal{A}_{\text{pre}}$ and $\mathcal{A}_{\text{onl}}$. They may share initial static state but cannot communicate during the game. In particular, $\mathcal{A}_{\text{onl}}$ does not see the input z nor $\vec{D}$ chosen by $\mathcal{A}_{\text{pre}}(z)$. It does obtain the inputs $pp, \sigma, \vec{aux}$. This restriction captures the fact that σ may encode a function of auxiliary information z in addition to the replicas of $\vec{D}$. Moreover, $\mathcal{A}_{\text{onl}}$ runs in parallel time at most T_{onl} whereas $\mathcal{A}_{\text{pre}}$ is only restricted to run in total time $O(poly(\lambda))$.

We will capture these security goals more formally through a game between a PoRep prover (adversary) and a challenger (Figure 3.3). A high level overview of the game is as follows. The prover/adversary is split into two components: a replication adversary and an online adversary whose sequential running time (i.e. parallel time) is bounded by a parameter T. The replication adversary is allowed to select the data inputs and the number of replicas k. The replicas do not need to be of all the same files. Specifically, we represent this input as a vector $D = (D_1, ..., D_k)$ of data files D_i each of size N, where some (or all) of the files are identical. The adversary also chooses k unique distinct identifiers for each file and k file tags. The prover can then compute on the data (as in phase 1 of replication) to produce an advice string σ for the online adversary as well as the auxiliary output $a\vec{u}x$, which includes N (the claimed size of each data file component). The online adversary receives a vector of k challenges from the challenger and must output k proofs. The challenger then runs the PoRep verifier on each and outputs the

---

[17]Mere existence of T_λ may seem like a weak requirement as T_λ could be arbitrarily large, however it is soundness that will fail in this case not correctness. For both soundness and correctness to hold there must exist some T_λ such that the honest prover can succeed in time T_λ and the cheating adversary cannot.

verifier's accept/reject decision.

k-replication   To capture "storing k independent replicas" we say that σ is a *k-replication* of a file D if the string σ can be partitioned into k substrings $\sigma_1, ..., \sigma_k$ such that each allow full recovery of the file D. Formally:

Definition 19. *Let $\mathcal{S}$ be a data source on $\{0,1\}^n$. A string σ is a k-replication of a data sample $D = (d_1, ..., d_k) \leftarrow \mathcal{S}^k$ if and only if there is a lossless randomized encoding scheme (enc, dec) and a partition of σ into k substrings $\sigma_1, ..., \sigma_k$ such that $\text{enc}(d_i) = \sigma_i$ and $\text{dec}(\sigma_i) = d_i$ for all $i = 1, ..., k$.*

Intuitively, security goal (1) can be captured by requiring that any online adversary in the PoRep game who is not given a *k-replication* of the data by the preprocessor adversary will fail with overwhelming probability. This would imply that if several provers each provide distinct PoReps of the same file then they are each dedicating unique resources[18] to storing the file. It would also imply that a prover who claims in a single proof to be storing multiple replicas of a file cannot physically deduplicate its storage. Unfortunately, this security property is impossible to achieve in a classical model of interactive computation (that does not include timing bounds on communication[19]), as we explain next. Instead, we will need to relax the security model to consider rational adversaries and bound the cost savings (measured in storage resources) of adversaries who do not store a *k-replication* of the data input. The security notion we call ϵ-rational replication could be informally interpreted as security against an "honest-but-opportunistic" adversary that will only employ a malicious strategy if they stand to save more than some ϵ cost doing so.

Data tag verification oracle   In order to be as general as possible we don't explicitly require τ_D to be a commitment to D. Instead, we assume the existence of an oracle $\mathcal{O}_{\text{check}}$ that implements a verification procedure on a data/tag pair, perhaps provided additional advice aux, i.e. on input (D, τ_D, aux) it returns $\mathcal{O}_{\text{check}}(D, \tau_D, \text{aux}) \rightarrow b \in \{0,1\}$. A special case is where τ_D is a commitment to D, aux is the opening hint, and $\mathcal{O}_{\text{check}}$ runs the commitment verification. In other scenarios $\mathcal{O}_{\text{check}}$ might verify signatures. As discussed previously, we allow for constructions that run in a private-key preprocessor mode and the data inputs/tags are provided by the preprocessor. For instance, they may be data authentication tags like in a more standard PoR scheme.

Definition 20. *PoRep-Exp$^{(\mathcal{A})}(z, pp) = (\hat{z}, \hat{b})$ is the PoRep game experiment (defined in Figure 3.3) with adversary $\mathcal{A} = (\mathcal{A}_{pre}, \mathcal{A}_{onl})$ on inputs z, pp from the environment, represented as a random variable where $\hat{z}$ is the output of $\mathcal{A}$ and $\hat{b}$ is the challenger's output. $\text{Out}_{\mathcal{A}}(pp, z) = \hat{z}$ is the random variable representing the first component of PoRep-Exp$^{(\mathcal{A})}(z, pp)$, and $\text{Succ}_{\mathcal{A}}$ is the probability that $\hat{b} = 1$.*

Definition 21 (Admissible PoRep adversary)**.** *For any $pp \leftarrow$ PoRep.Setup(λ, T) a PoRep game adversary $\mathcal{A} = (\mathcal{A}_{pre}, \mathcal{A}_{onl})$ is admissible for the experiment PoRep-Exp$^{(\mathcal{A})}(z, pp)$ if $\mathcal{A}_{pre}$ runs*

---

[18]The provers may be storing all the replicas on the same hard-drive, hence PoReps alone do not give a meaningful guarantee of fault-tolerant data storage.

[19]Consider a model with network communication round trip bounds and distance between parties. Two servers claim to be in two different locations and are each storing a replica of the same file. We could use distance bounding protocols combined with proofs of retrievability to verify the claim [52]

in $O(poly(\lambda))$ and $\mathcal{A}_{onl}$ runs in parallel time at most T. The adversary $\mathcal{A}$ is μ-admissible if additionally $\mathcal{A}_{pre}$ chooses data inputs $\vec{D}$ and tags $\vec{\tau}$ where for each i there exists aux_i such that $\mathcal{O}_{check}(D_i, \tau_{D_i}, aux_i) = 1$ and $Succ_{\mathcal{A}} \geq \mu$.

3.1.1 Impossibility of ideal replication security

Suppose that the PoRep adversary stores the replicas in a string σ. The adversary can then "sabotage" the replication by using say the first λ bits of σ as a key to encrypt the rest, and store the transformed string σ' that includes the λ bit key and ciphertext. Since the adversary can efficiently decode σ from σ' it will still pass the protocol with the same success probability (i.e. it *efficiently* decodes σ' and retrieves σ, and then follows whatever protocol behavior it would have initially on σ). Indeed, such "scrambling" attacks are impossible to prevent as there is always a trivially fast way to encode/decode one's state in a way that destroys the k-replication format. We state this formally in the following proposition. Our proof is in the random oracle model for simplicity, although an analogous statement can be proven in the plain model from symmetric key encryption (i.e. PRPs) as long as $|\sigma| > \lambda$ (this will always be true for $\mathcal{A}$ that succeeds on $|\vec{D}| > \lambda$, i.e. causes the challenger to output 1 with non-negligible probability).

Proposition 1. *For any admissible adversary $\mathcal{A} = (\mathcal{A}_{pre}, \mathcal{A}_{onl})$ there exists an admissible adversary $\mathcal{A}' = (\mathcal{A}'_{pre}, \mathcal{A}'_{onl})$ (with access to a random oracle H) such that for all pp, z and $\vec{D} = (D_1, ..., D_k)$ chosen by $\mathcal{A}_{pre}$ for any $k > 1$ the distributions $\mathsf{PoRep\text{-}Exp}^{(\mathcal{A})}(z, pp)$ and $\mathsf{PoRep\text{-}Exp}^{(\mathcal{A}')}(z, pp)$ are identical and for σ, σ' output by $\mathcal{A}_{onl}/\mathcal{A}'_{onl}$ respectively (in Figure 3.3, line 3), at most one of σ, σ' is a k-replication of $\vec{D}$ and yet $|\sigma'| = |\sigma|$.*

Proof. $\mathcal{A}'$ runs exactly as $\mathcal{A}$ with the following modifications. $\mathcal{A}'_{pre}$ runs $\mathcal{A}_{pre}$ until Line 2 of the game, obtaining $\sigma, a\vec{u}x$. If $|\sigma| = 1$ then $\mathcal{A}'_{pre}$ outputs $\sigma, a\vec{u}x$ and $\mathcal{A}'_{onl}$ on a 1-bit σ runs exactly like $\mathcal{A}_{onl}$. By hypothesis $k > 1$, hence a 1-bit σ is not a k-replication by definition. In case $|\sigma| > 1$, then let σ_0 denote the first bit of σ. $\mathcal{A}'_{pre}$ computes $H(\sigma_0) = r$, truncating the output of H such that $|r| = |\sigma| - 1$ (as $\mathcal{A}_{pre}$ runs in time $O(poly(\lambda))$ we may assume w.l.o.g. that the output of H is longer than σ) and sets $\sigma' = \sigma \oplus 0||r$. Note that only the first bit of σ' (i.e. σ_0) contains information about σ. All substrings of the remaining $|\sigma| - 1$ bits are uniformly distributed (they have zero mutual information with the remainder of σ). Therefore, assuming that σ is a k-replication of $\vec{D}$, then σ' is not a k-replication for $k > 1$. Finally, $\mathcal{A}'_{onl}$ receives σ', computes $r = H(\sigma_0)$ by reading the first bit of σ', and then recovers $\sigma = \sigma' \oplus 0||r$. Then $\mathcal{A}'_{onl}$ uses σ to run $\mathcal{A}_{onl}$. The output is identical to $\mathsf{PoRep\text{-}Exp}^{(\mathcal{A})}(z, pp)$. $\qquad\square$

3.1.2 Rational adversary model

Given the impossibility of achieving PoRep security entirely based on cryptographic guarantees, we turn instead to a framework of modeling attacks based on incentives. A *rational adversary* model has been used many times before in related works. In fact, it is implicit in proofs of space where security is reasoned via time/space tradeoffs [40] rather than challenge-response timing, which is necessary in proofs of space that are not parallel sound [3,46]. Storage enforcing commitments [28] were a precursor to PoR in which the prover must dedicate a minimum amount of storage in order to pass the protocol for a committed file F (preprocessed by a client with a secret key). The protocol does not guarantee that the prover is actually storing F, however a prover who does not would "waste" the space as it cannot use this space to store any other

"meaningful" data. Hourglass schemes [51] were proposed as a generic method to prove that a server is storing data encoded in a specified format, in particular for the use cases of encryption-at-rest and file watermarking. Proof of data replication can be viewed as a special case of this type of protocol, where the encoding is redundant. The system Mirror [7] considered proofs of data replication in a different setting, namely as a private-verifier PoR in which the server replicates the client's initially preprocessed file. Recall that this setting is fundamentally different from ours as the proofs rely on client preprocessing and are not publicly verifiable proofs of space. Location based storage proofs (LoSt) use distance bounding protocols to attempt to provide an even stronger claim that data is being stored in several different geographic locations [52]. The threat model considered in all of these works is an adversary that acts to minimize its costs, measured in storage or work (and combinations thereof).

In our setting, independent from any client setup/preprocessing, any prover that passes k independent PoRep interactive protocols on k data file commitments should not save any costs by running a strategy that does not store a recoverable copy of the file in each protocol, even if this involves storing duplicate copies of the same file. The same holds if the prover passes a single PoRep interactive protocol on a single committed data file that contains redundant data. More precisely, for any successful strategy there is another "well-behaved" and efficiently computable strategy satisfying k-replication that uses no more than a factor $1/(1 - \epsilon)$ more storage, i.e the adversarial strategy saves at most an ϵ fraction of the storage by deviating from honest k-replication. Importantly, this "well-behaved" strategy should still retain all the information that the original adversary was storing, which we capture by saying that the new strategy simulates the output distribution of the original adversary in the security game. We call this security notion ϵ-rational replication.

Security under composition A challenge that arises in both defining and proving ϵ-rational replication for PoReps is showing that adversarial strategies do not save costs even in composition with other protocols, in particular in the presence of auxiliary data on the server. While composition is always a security challenge often overlooked, it is especially salient here. One might think it would be sufficient to show that the adversary cannot compress its replicas by encoding them in a different (malicious) format that no longer preserves replication. However, we must also consider that the adversary may be able to use the replica storage to help store other useful information on its hard drive. Suppose, for instance, that the (incompressible) replicas could be used as a seed to compress this auxiliary data, yet in a way that also entangles the seed together with the auxiliary data, destroying the replication format (i.e. a weak compression Definition 5). In this case, the adversary may actually save costs by deviating.

Fisch and Silas [27] showed that proving this is likely impossible without strong knowledge assumptions. Their analysis focussed on the "weak compression attack" described above, where the PoRep storage could be used as a seed to compress auxiliary data in a way that also destroys the replication format. One might hope to prove rational security by showing that any adversary who weakly compresses auxiliary data in this way can be converted to a *better* adversary that compresses its auxiliary storage just as well without destroying its replicas, i.e. coverts the weak compression to a strong compression (Definition 6) that would save the seed independently in its original format. However, they prove that any black-box reduction of strong compression to weak compression for a given class of sources would yield a universal compression scheme for that class in the random oracle model. In other words, proving this is as hard as constructing a universal compression scheme that works on any source in the class.

"Knowledge-of-compression" assumptions   In light of their result, Fisch and Silas propose using a "knowledge of compression" assumption. Intuitively, any adversary who knows a way to compress auxiliary data z using an incompressible string s must only be taking advantage of the compressibility of z, hence entangling z together with s should not help. Specifically, the minimal assumption proposed in their paper is that any adversary who can compute a weak compression of z with an independent uniformly sampled seed s can also extract a strong compression of z. They also remark that some PoS constructions are proven secure by showing that the advice output s directly encodes part of a random oracle's function table, and in this case the minimal assumption suffices to show that any weak compression involving the PoS advice s as a seed can be converted to a strong compression. Unfortunately, this property is not true of PoReps because the underlying PoS advice actually encodes the committed data file D.[20] Therefore, we require a stronger knowledge of compression assumption that applies to any incompressible seed s, even where the incompressibility applies only to strictly time bounded compression schemes.

Definition 22 (Time bounded incompressibility). *A source $\mathcal{S}$ is (T_1, T_2)-incompressible to k bits if there does not exist (for any constant δ) a δ-lossy strong compression scheme (enc, dec) where enc runs in parallel time T_1 and dec runs in parallel time T_2 that compresses $\mathcal{S}$ to $k - \log(1/\delta)$ bits.*

Assumption 1 (Knowledge of strong compression). *Let $\mathcal{S}$ and $\mathcal{X}$ be any two samplable sources over $\{0,1\}^n$ such that $\mathcal{S}$ is (T_1, T_2)-incompressible to $(1 - \epsilon)n$ bits for some $\epsilon < 1$. For any compression scheme (enc, dec) that compresses $\mathcal{S} \times \mathcal{X}$ to m bits where enc runs in parallel time T_1 or dec runs in parallel time T_2, there exists a strong compression scheme (enc*, dec*) running in parallel time T_1 and T_2 respectively that compresses $\mathcal{X}$ to length $m - (1 - \epsilon)n$ bits sampling seeds from $\mathcal{S}$. Moreover, for any algorithm $\mathcal{A}$ that on inputs $\mathcal{S}$ and n outputs (enc, dec), there exists an efficient "extractor" which observes $\mathcal{A}$'s internal state and outputs (enc*, dec*).*

Definition of rational replication   We will fix several notations regarding discrete probability distributions before presenting our formal definition of ϵ-rational replication. If X is a 0/1 random variable with probability mass function p_X and Y is a discrete random variable on the same probability space then $p_X|Y : \mathrm{Sup}(Y) \to \mathbb{R}$ is the function $p_X|Y(w) = Pr[X = 1|Y = w]$. If f, g are two real valued functions on a set Ω then $f \succeq g$ if and only if $f(w) \geq g(w)$ for all $w \in \Omega$, i.e. f is an upper bound on g.

Finally, for any two random variables X, Y let $X(\lambda) \overset{\lambda}{\approx} Y(\lambda)$ denote that the distributions of X and Y are computationally indistinguishable.[21]

Definition 23 (Rational replication). *A PoRep construction is a μ-sound ϵ-rational replication if for any μ-admissible adversary $\mathcal{A} = (\mathcal{A}_{pre}, \mathcal{A}_{onl})$ there exists an admissible adversary $\mathcal{A}' = (\mathcal{A}'_{pre}, \mathcal{A}'_{onl})$ such that for all pp, z sampled by the environment and $\vec{D} = (D_1, ..., D_k)$ chosen by $\mathcal{A}_{pre}$, and for σ, σ' output by $\mathcal{A}_{onl}/\mathcal{A}'_{onl}$ respectively (in Figure 3.3, line 2) then for all z from the environment and $pp \leftarrow$ PoRep.Setup(λ, T) it holds that:*

---

[20] The security might still be proven by showing how to encode a random oracle table using S together with D [45], but the remark in [27] no longer applies in this case. Moreover, the prover can even choose D to be correlated with the auxiliary data z.

[21] The standard definition of computationally indistinguishable distributions with respect to a security parameter λ is that no poly(λ) time algorithms can distinguish samples from the two distributions except with probability negligible in λ.

1. $|\sigma| \le (1 - \epsilon)|\sigma'|$

2. σ' is a k-replication of $\vec{D}$

3. $\mathsf{Out}_{\mathcal{A}'}(pp, z) \stackrel{\lambda}{\approx} \mathsf{Out}_{\mathcal{A}}(pp, z)$ and $\mathsf{Succ}_{\mathcal{A}'}|\mathsf{Out}_{\mathcal{A}'}(pp, z) \succeq \mathsf{Succ}_{\mathcal{A}}|\mathsf{Out}_{\mathcal{A}}(pp, z).$[22]

Necessary and sufficient conditions for ϵ-rational replication PoReps satisfying ϵ-rational replication are implicitly proofs of space with an ϵ space gap. The converse is not necessarily true, i.e. ϵ-tight proofs of space are not immediately PoReps satisfying ϵ-rational replication. Furthermore, unless τ_D is *binding* to D relative to $\mathcal{O}_{\mathsf{check}}$, meaning that for any τ_D there does not exist (or it is computationally difficult to produce) $D' \ne D$ and $\mathsf{aux}', \mathsf{aux}$ such that $\mathcal{O}_{\mathsf{check}}(D, \tau_D, \mathsf{aux}) = \mathcal{O}_{\mathsf{check}}(D', \tau_D, \mathsf{aux}') = 1$, then it is unclear how a scheme could satisfy our definition (or any meaningful definition of replication security for that matter where the adversary is allowed to choose the data). If $\mathcal{O}_{\mathsf{check}}$ is not binding then even rational replication security would likely fail because the adversary could find an alternative data input $D' \ne D$ consistent with the same tag and run the rest of the protocol successfully on D' as if it were the input. Formally, this would fail our security definition if any adversary that stores a k-replication of the original data input D could not output the same distribution (e.g. the adversary's output $\hat{z}$ is some function of z and D').

Implicit PoS protocol If a PoRep construction satisfies Definition 23 then PoRep.Setup, PoRep.Replicate, PoRep.Prove, and PoRep.Verify implicitly define a secure PoS as follows:

- Setup runs PoRep.Setup(λ, T)

- Initialization The prover runs $R, \mathsf{aux} \stackrel{\mathrm{R}}{\leftarrow} \mathsf{PoRep.Replicate}(id, \tau_D, \tilde{D})$ and outputs $\tau_D = \mathsf{aux}$ and $S = R$.

- Execution The verifier's challenge is $r \leftarrow \mathsf{PoRep.Poll}(\mathsf{aux})$ and the prover responds with the proof $\pi \leftarrow \mathsf{PoRep.Prove}(R, \mathsf{aux}, id, r)$. The verifier runs $\mathsf{PoRep.Verify}(id, \tau_D, r, \mathsf{aux}, \pi)$.

We say that a PoRep scheme is an (s, t, μ)-sound PoS if its implicit PoS protocol is an (s, t, μ)-sound PoS (see Section 2.7, or Definition 17).

Lemma 3 (PoS is necessary). *If a PoRep is a μ-sound ϵ-rational replication then it is an $((1 - \epsilon)kN, T, \mu)$-sound PoS.*

Proof. Suppose not, i.e. there is some PoS adversary that succeeds in passing verification with probability greater than μ using only $(1 - \epsilon)kN$ storage.

Let $\vec{D}$ be a vector of k identical copies of an incompressible data input D. By incompressible, we mean that for any randomized enc/dec scheme such that $Pr[\mathsf{enc}(\mathsf{dec}(D)) = D] = \xi$ then $|\mathsf{enc}(D)| \ge N - \log(1/\xi)$ where $N = |D|$. It follows that any k-replication of D must be at least kN bits long because by definition it can be partitioned into k substrings each of which are a deterministic encoding of D. Now take the PoS adversary $\mathcal{A}_{PoS}$ who is able to pass Execution within time T with $(1 - \epsilon)kN$ storage and construct $\mathcal{A}_{PoRep} = (\mathcal{A}_{\mathrm{pre}}, \mathcal{A}_{\mathrm{onl}})$. $\mathcal{A}_{\mathrm{pre}}$ simulates the Initialization phase for $\mathcal{A}_{PoS}$ to obtain the outputs (Φ, S). It then sets $\mathsf{aux} = \Phi$ and $\sigma = S$.

---

[22]This says that for any value of $\hat{z}$ the probability $\mathcal{A}$ succeeds given that it outputs $\hat{z}$ is bounded by the probability that $\mathcal{A}'$ succeeds given that it outputs $\hat{z}$.

Then for each challenge query, $\mathcal{A}_{\text{onl}}$ simulates the Execution phase for $\mathcal{A}_{PoS}$, passing it Φ and S. Thus, by assumption $\mathcal{A}_{\text{onl}}$ succeeds (i.e. the challenger outputs $\hat{b} = 1$) with probability greater than μ. In other words, $\mathcal{A}_{PoRep}$ is μ-admissible. Now, since $|\sigma| \leq (1 - \epsilon)kN$ it cannot be a k-replication of D due to the fact that D is incompressible and any k-replication of D must be at least kN bits long. In fact, if any $(\mathcal{A}'_{\text{pre}}, \mathcal{A}'_{\text{onl}})$ outputs σ' where σ' is a k-replication of D then $|\sigma'| \geq kN > |\sigma|/(1 - \epsilon)$. On the other hand, by the hypothesis that the PoRep is a μ-sound ϵ-rational replication, there exists some $(\mu - \mathsf{negl}(\lambda))$-admissible $\mathcal{A}'$ that outputs a k-replication σ' (Condition 3 + Condition 2) such that $|\sigma'| < |\sigma|/(1 - \epsilon)$ (Condition 1). This is a contradiction. $\qquad\square$

Implicit PoRC protocol If the data tag τ_D is binding to the data input D and $\vec{\text{aux}}$ is a binding commitment to the replica R then a PoRep may implicitly define a $(\delta, \mathcal{C})$-PoRC protocol of (D, R) with respect to (τ_D, aux) (Section 2.4). The cover $\mathcal{C}$ in this case is the set of pairs of blocks (R_i, D_i) such that R is defined to be the correct output of $\mathsf{PoRep.Replicate}(id_i, \tau_D, D)$, D_i is the ith block of D, and R_i is the ith block of R. The setup just runs the PoRep setup and outputs parameters pp. The $\mathsf{PoRC.Prove}$ and $\mathsf{PoRC.Verify}$ procedures simply run the PoRep prover and verify procedures.

- $\mathsf{PoRC.Commit}(pp, D)$: Run $\mathsf{PoRep.Preproc}(sk, D)$ to obtain the data tag τ_D along with $\tilde{D}$. Then run $\mathsf{PoRep.Replicate}(id, \tau_D, \tilde{D})$ to get (R, aux). Save the hint $\Lambda = (R, id)$. Output the commitment $com = (\tau_D, \text{aux})$.

- $\mathsf{PoRC.Open}(pp, com, \Lambda)$: Parse $\Lambda = (R, id)$ and $com = (\tau_D, \text{aux})$. Next run the extraction procedure $\mathsf{PoRep.Extract}(pp, id, \tau_D, R)$ to obtain D. Query $b \leftarrow \mathcal{O}_{\mathsf{check}}(D, \tau_D, \text{aux})$. If $b = 0$ output $\perp$ otherwise output (D, R).

- $\mathsf{PoRC.Prove}(pp, com, \Lambda, c)$: Parse $\Lambda = (R, id)$ and $com = (\tau_D, \text{aux})$. Generate the PoRep proof π' as $\mathsf{PoRep.Prove}(R, \text{aux}, id, r)$ output the PoRC proof $\pi = (\pi', id)$.

- $\mathsf{PoRC.Verify}(pp, com, c, \pi)$: Parse $com = (\tau_D, \text{aux})$ and $\pi = (\pi', id)$. Output the result b of $\mathsf{PoRep.Verify}(id, \tau_D, c, \text{aux}, \pi')$.

We say that a PoRep scheme is a μ-sound δ-PoRC if its implicit $(\delta, \mathcal{C})$-PoRC protocol as just described is μ-sound (see Definition 12).

The following powerful lemma establishes criteria that along with Assumption 1 are sufficient for ϵ-rational replication. This greatly simplifies the analysis for any PoRep construction that is also implicitly a secure public-coin PoRC protocol.

Lemma 4 (Sufficient conditions for rational replication). *Given Assumption 1, if a PoRep construction with setup parameters T, λ satisfies the following conditions for some integer c:*

  (a) It is correct (see Definition 18)

  (b) It is a μ-sound $(1 - \delta)$-PoRC, and the parallel runtime blowup of the extractor is less than a factor c, i.e. it is PRAM strongly μ-sound (see Definition 12).

  (c) k independent PoReps are a parallel $((1 - \epsilon + \delta)kN, (c+1)T, \mu)$-sound PoS (see Section 2.7, or Definition 17).

then the PoRep construction is a μ-sound ϵ-rational-replication.

Proof. The outline for the proof is as follows. We consider any μ-admissible PoRep adversary $\mathcal{A} = \mathcal{A} = (\mathcal{A}_{\text{pre}}, \mathcal{A}_{\text{onl}})$ and construct the "well-behaved" adversary $\mathcal{A} = (\mathcal{A}'_{\text{pre}}, \mathcal{A}'_{\text{onl}})$ that satisfies the three conditions of Definition 23. Since $\mathcal{A}$ is μ-admissible for each i it chooses each D_i, τ_{D_i} and outputs aux_i such that $\mathcal{O}_{\text{check}}(D_i, \tau_{D_i}, \text{aux}_i) = 1$. A μ-admissible PoRep adversary is also a μ-admissible PoRC adversary in the implicit $(1 - \delta, \mathcal{C})$-PoRC protocol. Hypothesis (b), that the implicit $(1 - \delta, \mathcal{C})$-PoRC protocol is μ-sound, by definition means there exists a public extraction algorithm $\mathcal{E}_{\mathcal{A}}$ that $\mathcal{A}'_{\text{onl}}$ could run with $\mathcal{A}_{\text{onl}}$ to extract at least a $(1 - \delta)$ fraction of the correct blocks in each (R_i, D_i), denoted by (R_i^*, D_i^*). Let $R^* = (R_1^*, ..., R_k^*)$ and $D^* = (D_1^*, ..., D_k^*)$. $\mathcal{E}_{\mathcal{A}}$ runs in time at most $c \cdot t_{\text{onl}}$ where $t_{\text{onl}} < T$ is the runtime of $\mathcal{A}_{\text{onl}}$ (by the assumption that $\mathcal{A}$ is admissible).

We show (in Subclaim 1) that $\mathcal{A}_{\text{pre}}, \mathcal{A}_{\text{onl}}$, and $\mathcal{E}_{\mathcal{A}}$ can be used to construct a lossless compression scheme (enc, dec) that encodes $\mathcal{E}_{\mathcal{A}}$'s output R^* and $\mathcal{A}$'s output $\hat{z} = \text{Out}^{\mathcal{A}}(pp, z)$ in $\text{PoRep-Exp}^{(\mathcal{A}}(pp, z)$, where $\text{enc}(R^*, \hat{z}) = (\sigma^*, \text{aux})$ such that $|\sigma^*| \leq |\sigma| + \text{negl}(\lambda)$ and dec runs in total time cT. On the other hand, we show (in Subclaim 2) that Hypothesis (c) implies that R^* is $(\text{poly}(N), (c+1)T, \mu)$-incompressible to $(1-\epsilon)kN$ bits. Together with Assumption 1, Subclaim 1 and Subclaim 2 imply that $\mathcal{A}'_{\text{onl}}$ can compute a new strong compression scheme $(\text{enc}^*, \text{dec}^*)$ that uses R^* as a seed to encode compress $\hat{z}$ to $|\sigma^*| - (1 - \epsilon)kN \leq |\sigma| - (1 - \epsilon)kN + \text{negl}(\lambda)$ bits. Finally, $\mathcal{A}'_{\text{pre}}$ outputs $\sigma' = R||enc^*(R^*, \hat{z})$ and the same $\vec{\text{aux}}$, and $\mathcal{A}'_{\text{onl}}$ uses dec^* to recover $\hat{z}$. Note that $|\sigma'| \leq |R| + |\sigma| - (1 - \epsilon)kN + \text{negl}(\lambda) = |\sigma| + \epsilon kN + \text{negl}(\lambda)$. $\mathcal{A}'_{\text{onl}}$ finally runs the honest PoRep.Prove so $\text{Succ}_{\mathcal{A}'} = 1$ and $\text{Out}_{\mathcal{A}'} \approx \text{Out}_{\mathcal{A}}$ are identically distributed.

The existence of this $\mathcal{A}'$ for any μ-admissible adversary $\mathcal{A}$ shows that the PoRep satisfies μ-sound ϵ-rational replication. It satisfies $\text{Out}^{\mathcal{A}'} \approx \text{Out}_{\mathcal{A}}$ and $\text{Succ}\mathcal{A}'|\text{Out}_{\mathcal{A}'} \succeq \text{Succ}_{\mathcal{A}}|\text{Out}_{\mathcal{A}}$. The the advice σ' includes R and is thus a k-replication of D (Condition 2) with $|\sigma'| \leq \epsilon kN + |\sigma| + \text{negl}(\lambda) \leq \frac{\epsilon}{1-\epsilon}|\sigma| + |\sigma| + \text{negl}(\lambda) \leq |\sigma|/(1 - \epsilon) + \text{negl}(\lambda)$ (Condition 1).

Proof of Subclaim 1: Let $\mathcal{R}$ denote the distribution of R^* extracted by $\mathcal{E}_{\mathcal{A}}$ over the randomness of $\mathcal{E}_{\mathcal{A}}, \mathcal{A}_{\text{pre}}$, and $\mathcal{A}_{\text{onl}}$. Note that because $\mathcal{A}$ is μ-sound R^* always contains a $1 - \delta$ fraction of the blocks of R, where R is the correct output on the input D chosen by $\mathcal{A}_{\text{pre}}$ except with at most probability $\text{negl}(\lambda)$ (that either the binding of $\mathcal{O}_{\text{check}}$ is broken or the extraction fails). Let $\mathcal{Z}$ denote the distribution of $\hat{z}$ given the fixed environment inputs z, pp. By simulating the game for $\mathcal{A}$ and running the extractor $\mathcal{E}_{\mathcal{A}}$ we can perfectly sample both of these distributions. The sampled values R^* and $\hat{z}$ are given as input to enc along with aux and the randomness ρ used in the sampling. The randomness ρ and aux will be shared by both enc and dec so we treat this as a seed.

- enc uses ρ to deterministically rerun $\mathcal{A}_{\text{pre}}$ to obtain σ and sets $\text{enc}(\rho, R^*, \hat{z}) = \sigma$.

- dec runs $\mathcal{E}_{\mathcal{A}}$ with inputs σ and $\vec{\text{aux}}$ to extract R^* from $\mathcal{A}_{\text{onl}}$ (in parallel time at most $c \cdot t_{\text{onl}}$) and outputs R^* while also using ρ to rerun $\mathcal{A}_{\text{onl}}$ on a given challenge vector to obtain $\hat{z}$. dec outputs R^* and $\hat{z}$. Hence, (enc, dec) is a strong compression scheme on the source $\mathcal{R} \times \mathcal{Z}$ with loss negligible in λ, i.e. $Pr[\text{dec}(\rho, \text{enc}(\rho, R^*, \hat{z}) = R^*, \hat{z}] > 1 - \text{negl}(\lambda)$. Finally, by Fact 1, it can be converted to a lossless compression with a $\text{negl}(\lambda)$ increase in output length.

Proof of Subclaim 2: As R is the correct output of PoRep.Replicate and the PoRep construction is a $((1 - \epsilon + \delta)kN, (c + 1)T, \mu)$-sound PoS, R is $(\text{poly}(N), (c + 1)T, \mu)$-incompressible to $(1 - \epsilon + \delta)kN$ bits. Otherwise, for some $\mu' > \mu$ an adversary could compress R during the

replication phase to $(1 - \epsilon + \delta)kN$ bits and store this μ'-lossy compression as its advice. It could then recover R in the challenge-response phase with probability μ' and run the honest PoRep.Prove in total parallel time less than $(c + 1)T$. This adversary would break the presumed PoS soundness. Likewise, as R^* consists of a $1 - \delta$ fraction of blocks from R, it is $(\text{poly}(N), (c + 1)T, \mu)$-incompressible to $(1 - \epsilon)kN$ bits, because compressing R^* to $(1 - \epsilon)kN$ bits is equivalent to compressing R to $(1 - \epsilon + \delta)kN$ bits.

$\square$

Remark on δ loss: Lemma 4 requires that the PoRep has only a $\epsilon - \delta$ space gap as PoS in order to prove it satisfies ϵ-rational-replication, where δ is the loss in the PoRC. The reason this affects the proof that the PoRep is ϵ-rational-replication is entirely due to the environment auxiliary input z and the need to show that the adversary $\mathcal{A}'$ can still output the same distribution (including the function of z) as the adversary $\mathcal{A}$. (This necessitated constructing $\mathcal{A}'$ that extracts the replica R from $\mathcal{A}$). If this requirement were dropped (or the auxiliary input z is empty) then conditions (a) and (c) suffice without any δ loss, i.e. if the PoRep is correct and a $(1 - \epsilon)kN, T)$-sound PoS then it is ϵ-rational secure. The reason is that $\mathcal{A}'$ could just run the honest replication strategy, where $\sigma' = R$ and $|R| = kN$. Any adversary that causes the challenger to output $\hat{b} = 1$ must use σ of size at least $(1 - \epsilon)kN$ by the PoS soundness.

What is missed by simply composing PoR and PoS? It is instructive to consider simply combining a proof of retrievability (PoR) with a proof of space (PoS) as a trivial candidate PoRep. For a file D of size N, the honest prover would use N bits of storage to produce a PoS and separately store D. First, this results in at least a factor two *space gap* (the gap between the honest prover's total space usage and the lower bound on an adversary's space). The more criticial issue is that the adversary is clearly incentivized to only use N bits of storage for the PoS and drop the file entirely, provided that it can recover the file from another source. *There is explicitly a factor two added cost for storing the file in addition to producing a proof of space.* A prover is strongly incentivized to deduplicate its storage. In other words, this protocol fails ϵ-rational replication for any $\epsilon < 1$, nor does it have data-independent costs. Thus we can see that at a minimum our definition requires that PoReps are simultaneously a PoS and PoR without costing the miner any extra space to store the entire file (or some encoding thereof that allows full recovery).

Space utilization There is still a silly way to combine PoS and PoR to achieve ϵ-rational replication without imposing another efficiency requirement on the construction that has to do with *space utilization*. Suppose a PoS of N bits guarantees that the adversary is using at least δN bits (while the honest prover uses N bits). The trivial PoRep construction on a file D of size N would be a δ-PoS on $m >> N$ bits (say $m \geq kN$) combined with a PoR of D. The naive prover following this protocol uses at least $(k + 1)N$ bits, while an adversary must use at least δkN bits. The additional cost to this adversary of storing D is at most N bits, for a total storage of $(\delta k + 1)N$, which is an $\epsilon = 1/(\delta k)$ overhead. This can be made arbitrarily small by increasing k. More generally, we can modify any PoRep scheme that is both a PoR and a δ-PoS to operate on padded data $D||0^m$ for $m = k|D|$ so that there is ony an $\epsilon = 1/(\delta k)$ cost to additionally storing the file. However, at the same time, this generic construction utilizes only an ϵ fraction of the prover's total space for useful storage. Ideally, we would like a construction that utilizes most of the space, independent of ϵ. Restricting to PoRep constructions that can

be tuned to satisfy ϵ-rational replication using $O(N)$ space even as $\epsilon \to 0$ excludes these wasteful constructions.

Private-verifier setting We note that these subtleties in defining proofs of replication are also relevant in the private-verifier setting of proofs of data replication. They were missed in the security model proposed by Mirror [7], whose definition is in fact satisfied by even the trivial combination of a PoS and PoR. Mirror's security definition compares the adversary's storage to the storage of a prover who stores both the original file D and r replicas specified by the protocol. Consider the trivial construction in which each "replica" is simply an independent proof of space initialization, and the verification procedure involves a PoR of D along with r PoS verifications. Since in this setting the data file is chosen by the verifier/challenger and is assumed to be incompressible, the adversary must use as much space as the honest prover in order to pass verification.[23]

3.1.3 Security for decentralized storage networks

As we touched upon in the introduction, a primary intended use-case for PoReps is as a protocol that will help both incentivize and monitor file storage in a distributed network of nodes, which are compensated by the network in return for storing client data. Critically, these nodes are not directly compensated by the client. Rather, they are compensated for storing any data as long as the network verifiers (distinct from clients submitting data) can publicly verify they are storing the data. In fact, the network protocol may aggregate data pieces from several clients together and pass that to a storage node.

If clients were to pay storage nodes directly, say by putting a lump sum of money in escrow that the storage node receives in installments for continuing to store the file, then PoReps aren't necessary. The role of network verifiers would be to verify that the storage node is still storing the client's file using a standard proof of retrievability with outsourced verification, and then approve transfer of installment funds from the escrow account to the storage node. The storage node has nothing to gain from colluding with the client. It is only when we want the network itself to incentive storage participation from a source external to the client/node that PoReps actually become relevant.

The main thesis of the Filecoin [1] protocol is that by double using PoReps as a "proof-of-resource" (i.e in place of proof-of-work) and proof-of-storage both the network and the client simultaneously benefit.[24] The network rewards the storage miners by printing inflationary rewards (similar to mining rewards for Bitcoin proofs-of-work miners), thereby subsidizing the cost of file storage. Notice how this application of PoReps as a "useful" proof-of-storage to mine Filecoin and subsidize storage costs is extremely sensitive to the security property of PoReps. Consider a PoRep such as the naive combination of PoS and PoR, which does not satisfy ϵ-replication (for $\epsilon < 1$). A miner would actually double its profits in Filecoin mining rewards by deduplicating storage because it could now use the extra storage space to generate twice as many proofs of storage. Morever, miners could double their Filecoin rewards by producing PoReps for useless files that the prover generates from a small seed instead of storing real client data (called the "generation attack" in [2]. This means that Filecoin miners will charge even more to use their space to store user files, raising rather than subsidizing the cost of storage. More generally, if the PoReps do not have data-independent costs then the Filecoin mining rewards could have

---

[23]It satisfies Mirror's definition of δ-binding as long as the PoS has at most a $1 - \delta$ space gap.

adverse effects on storage costs, driving miners to increase the cost of some files while lowering the cost of others.
.

4   Constructions

In all of the constructions we describe in this report we will skip the description of PoRep.Preproc. The data is preprocessed in one of the modes described, and we start with the preprocessed data $\tilde{D}$ and data tag τ_D. We also assume there is an external verification procedure that the verifier may query on any block d_i of the file $\tilde{D}$, its position i, and τ_D, which returns a result that we denote by $\mathcal{O}_{\mathsf{check}}(d_i, i, \tau_D) \to b \in \{0,1\}$.

Building blocks   All constructions use a VDE scheme $\{\mathsf{VDE.Setup}, \mathsf{VDE.Enc}, \mathsf{VDE.Dec}\}$ with identical input space and code space over $\{0,1\}^m$ for some $m = O(\lambda)$ determined by the VDE setup, as well as a hash function $H : \{0,1\}^* \to \{0,1\}^m$ modeled as a random oracle (i.e. mapping strings of arbitrary length to strings of length m).

The construction DRG-PoRep uses $\mathsf{DRG.Sample}(n, d)$, which outputs an $(\alpha n, \beta n, d)$ depth robust graph with probability $1 - \mathsf{negl}(n)$ for $d = O(\log n)$. In the Section 5 we discuss instantiations of DRG.Sample and how d can be chosen to target different values of the constants $0 < \beta < \alpha < 1$ (both analytically and empirically given our implementation of the best known attacks). In particular, in DRG-PoRep the value of α must be very small, as this will determine the ϵ-rational security of the scheme (in fact $\epsilon = \alpha$).

The latter two constructions also use a vector commitment scheme with algorithms VC.Setup, VC.Com, VC.Open, and VC.Verify.

Security parameters   Each PoRep scheme will take two security parameters λ, κ as input. The parameter λ determines the overall security level of the construction (e.g. 80-bit, 128-bit, 256-bit), whereas the parameters κ determines the soundness of the "online" proofs generated by PoReP.Prove. Since these are run repeatedly in a challenge-response protocol, we leave open the possibility that certain applications may target a non-negligible online soundness error such as $1/1,000$, $1/100$ or even $1/3$. Each challenge-response proof will be subject to this soundness bound independently thus security is amplified against a prover that tries to cheat many times. We will discuss how concrete choices for each of the schemes impact security/efficiency.

4.1   Basic PoRep from Verifiable Delay Encodings

A very basic PoRep construction is simply to use the VDE scheme to derive a unique replica R_{id} as a *slow encoding* of the file F on a given id. This scheme was already sketched in [14],

---

[24]Decentralized transactional networks that follow the Nakamoto paradigm for consensus, i.e. proof-of-work blockchains, require consensus nodes to produce some form of "proof-of-resource" to ensure Sybil resistance. Depending on its instantiation, this proof-of-resource may also contribute global costs to transacting by adding an expense to transaction processing. Often these systems reduce the local impact of this cost on transaction fees by printing an inflationary reward that goes directly to the consensus node processing the transaction (although the inflation silently taxes the economy). This has the effect of subsidizing the cost of transacting, the same way that governments may use inflation to subsidize food or economic activity to stimulate an increase in GDP. It has been observed that the inflationary reward is more than just a subsidy mechanism, as it is actually critical to the security of Nakamoto consensus [39]

which generalized a scheme originally proposed by Sergio Demian Lerner [33], although neither works provided a formal security analysis. During the challenge-response period, the verifier periodically checks that the server has been persistently storing the encoding R_{id} by challenging the server for R_{id} itself. If the delay parameter of the VDE scheme is set appropriately relative to the verifier's polling period, then a prover that has deleted the replica R_{id} at any point during the polling period will not be able to re-derive the encoding in time to respond to the verifier's challenge. We also need to ensure that the replicas on two different ids are independent, and cannot be derived from one another more quickly than deriving each from scratch. The relevant security property of a VDE scheme is that on any independent *random* challenge it is sequentially hard to compute, even given a polynomial number of prior encodings of other points. We can enforce the independence of inputs by using a random oracle[25] H applied to the replica id. Specifically, instead of encoding F directly, we encode $F \oplus H(id)$. Recovery of F is still possible as id is known to the decoder. Technically, as F may consist of multiple blocks of length m we will encode each ith block individually XORed with $H(i||id)$. The verifier only needs to challenge for a randomly sampled constant number of the encoded blocks (and check them individually) in order to ensure that no more than an ϵ fraction have been deleted.

Lerner's initial proposal of this protocol used a Pohlig-Hellman cipher as a weak instantiation of the VDE and also focussed on the special use-case of proving unique storage of a copy of a fixed file known to all the verifiers (it was proposed as a way to prove unique storage of a copy of a blockchain). In our case, we need to adapt the protocol to handle data files specified only by their commitment tag τ_D that may be initially unknown to the verifier. The PoRep needs to still be a PoS in this case. Fundamentally, for this to be secure the slow replica encoding must depend on this tag. Otherwise, the prover can first set the PoRep replica to any arbitrary value and then run the data extraction algorithm on a particular id to find data consistent with this replica. This requires only a simple modification which is to use both id and τ_D as input to the random oracle H.

4.1.1 Basic-VDE-PoRep

PoRep.Setup$(\lambda, T) \to pp$ Set $\Delta = 2T$ and run VDE.Setup$(\Delta, \lambda) \to pp_{\text{vde}}$. This specifies the block length m, and provides implicit input parameters to VDE.Enc and VDE.Dec. Lastly the setup specifies an integer $\ell = \ell(\lambda)$ of size $O(\lambda)$.

PoRep.Replicate$(id, \tau_D, \tilde{D}) \to R, aux$ Parse $\tilde{D}$ as a file of N blocks $d_1, ..., d_N$ each a string in $\{0,1\}^m$. For each i compute $R_i = \text{VDE.Enc}(d_i \oplus H(id||\tau_D||i))$. Output $R = (R_1, ..., R_N)$ and $aux = N$.

PoRep.Extract$(id, \tau_D, R) \to \tilde{D}$ Parse $R = (R_1, ..., R_N)$ and for each i compute $d_i = \text{VDE.Dec}(R_i) \oplus H(id||\tau_D||i)$. Output $\tilde{D} = (d_1, ..., d_N)$.

PoRep.Poll$(N) \to r$ For $i = 1$ to ℓ randomly sample $r_i \xleftarrow{\text{R}} [N]$. Output $r = (r_1, ..., r_\ell)$.

PoRep.Prove$(R, N, id, r) \to \pi$ Parse $R = (R_1, ..., R_N)$ and $r = (r_1, ..., r_\ell)$. Output the proof $\pi = (R_{r_1}, ..., R_{r_\ell})$.

---

[25]The random oracle can be replaced with a collision-resistant hash function if only a pebbling adversary is considered in the security analysis.

PoRep.Verify$(id, \tau_D, r, N, \pi) \rightarrow 0/1$ Parse the proof $\pi = (\pi_1, ..., \pi_\ell)$ as ℓ strings in $\{0,1\}^m$. For each $i = 1$ to ℓ do:

1. Compute $\hat{d}_i = \mathsf{VDE.Dec}(\pi_i) \oplus H(id||r_i)$

2. Query $b_i \leftarrow \mathcal{O}_{\mathsf{check}}(\hat{d}_i, r_i, \tau_D)$

If $b_i = 1$ for all i then output 1 (accept), otherwise output 0 (reject).

VDE instantiation   We can instantiate the basic PoRep construction with the Sloth [32] VDE. With a block size of $m = 4096$ and time delay T of 10 minutes the Sloth decode takes approximately 0.15 seconds. For the same target delay with $m = 512$ decoding takes approximately 3 seconds. These were measured on a 2.3 GHz Intel Core i7.

Replication time   For a target polling period of 5 minutes, choosing the block size gives a tradeoff between proof size and initialization time. With a block size of $m = 4096$ and time delay T of 10 minutes, replication of files up to 50KB ($N = 100$) takes approximately 1 hour on 16 parallel cores. With block size $m = 256$ we can only support files up to 320 bytes for replication under 1 hour. If instead we fix the file size (e.g. up to 50KB) but decrease block size by a factor κ, then we both increase initialization time by a factor κ and decrease proof size by a factor κ.

Proof size   The proof size is a constant ℓm bits independent of the file size. To ensure (with soundness error δ) that no more than an ϵ fraction of the blocks have been deleted the number of queries ℓ should be set to $\ell = \log(\delta)/\log(1-\epsilon)$. For example to detect deletion of 1% of the data blocks with soundness error $1/3$ we would require $\ell \approx 100$, in which case the proof might as well include the entire replica for $N < 100$. To detect deletion of 5% with soundness $1/3$ only requires $\ell \approx 20$. For 80% we can set $\ell \approx 5$.

Correctness and Security   Correctness is immediate from the construction and correctness of the underlying primitives. The construction's rational security is optimally tight, i.e. it satisfies μ-sound ϵ-rational-replication for any arbitrary $\epsilon < 1$ and $\mu = \mathsf{negl}(\lambda)$. We provide the formal analysis in Section 6 (Theorem 3). The construction is also therefore a tight[26] (Lemma 3).

Erasure code preprocessing   In combination with an optimal rate $1 - \epsilon$ erasure code during preprocessing then it becomes perfectly rational for the prover to store a recoverable encoding of the file because even if it deletes an ϵ fraction of the blocks it will still be perfectly recoverable. Likewise, if the prover generates k independent replicas of a file D then because it uses $(1-\epsilon)kN$ space overall it can store ϵN blocks of each replica so that D is still recoverable from each. Increasing the tolerance of the erasure codes results in an increase in replica size, but may justify reducing the tightness of the PoS/rational-replication security (i.e. reduce ϵ). For example, on $N = 100$ blocks, setting $\epsilon = 0.01$ and using an erasure code that recovers from 1% deletion results in only a 1% increase in file size but reducing a proof that is equal in size to the original

---

[26]Although the VDE based PoRep is a concretely tight proof of space with practical proof sizes, unlike other proofs of space in the literature it does not have an efficient execution time, which scales quadratically as $O(NT) = O(N^2)$ where T is the time delay parameter.

Data File		Replica
$\tilde{D}_1$	$\xrightarrow{\mathsf{VDE.Enc}(\tilde{D}_1 \oplus H(id\|\|i))}$	R_1
$\tilde{D}_2$	$\xrightarrow{\mathsf{VDE.Enc}(\tilde{D}_2 \oplus H(id\|\|i))}$	R_1
$\vdots$	$\vdots$	$\vdots$
$\tilde{D}_N$	$\xrightarrow{\mathsf{VDE.Enc}(\tilde{D}_N \oplus H(id\|\|N))}$	R_N

Figure 4.1: Illustration of PoRep.Replicate in the basic PoRep construction using a VDE.

file. On the other hand if $\epsilon = 0.20$ and D is preprocessed with an erasure code that tolerates 20% deletion, then we trade an increase in the replica size by 20% for a proof size of only 6% of the original file size.

4.2 Block Chaining Encodings

Clearly the Basic-VDE-PoRep is impractical for large file sizes. A 1 GB size file with block size of 512 bytes would take over 13 days to replica on a machine with limited parallelism. Increasing the block size to reduce replication time impacts proof size and is also limited by the message space of the VDE scheme. VDE schemes like Sloth operate on relatively small input spaces as larger input spaces are more susceptible to parallelization attacks. The fundamental issue in the Basic-VDE-PoRep construction is that the VDE (tuned for the polling period delay T) is applied individually to each block of the file thus allowing a fully parallel attacker to derive the entire replica within time T whereas it takes a non-parallel prover take time TN. One might consider relaxing the security requirement to preclude excessive parallel attacks and reduce the time delay T required to encode each block. However, any attacker who uses only a factor k more parallelism will be able to reduce replication time by a factor k. Thus, for any reasonable factor k advantage (over honest provers) that an attacker may achieve, the time to replicate for the honest provers be at least a factor k times longer than the polling period. Furthermore, with the current verification strategy of randomly sampling ℓ blocks to challenge, the prover can re-derive the specific challenged blocks in total time ℓT (or parallel time T with only ℓ parallelism).

A natural way to reduce the overall replication time while maintaining the sequential hardness is to *chain* the encodings of each block, similar to encryption in block chaining cipher modes. A simple chaining would modify PoRep.Replicate in the Basic-VDE-PoRep by deriving from each R_i (encoding of block d_i) a key $k_i = H(id\|\|R_i)$ to be used in the encoding $R_{i+1} = \mathsf{VDE.Enc}(d_{i+1} \oplus k_i)$ (of block d_{i+1}) as shown in Figure 4.2. Each R_i can still be decoded locally given only R_{i-1} as $D_i = \mathsf{VDE.Dec}(R_i \oplus k_i)$ where $k_i = H(id\|\|R_{i-1})$. We would then reduce the time delay T for each call to VDE.Enc such that $T \cdot N$ is equal to the desired replication time.
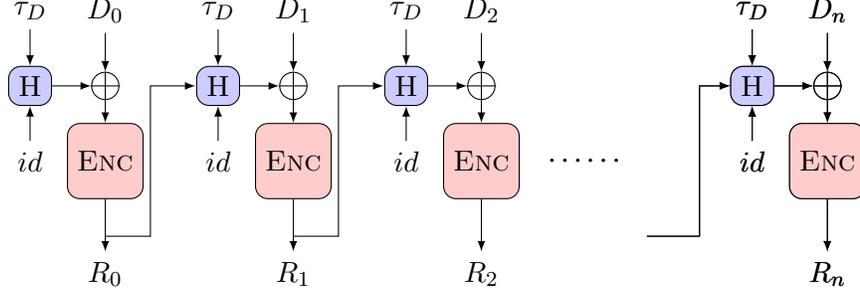
Figure 4.2: Basic-VDE-PoRep in CBC-mode (insecure).

The problem with this basic chaining method is that it has a smooth time/space tradeoff. An adversarial prover can store only each kth block (reducing overall storage by a factor k) and yet it can recompute any individual block with only k calls to VDE.Enc. With sufficient parallelism it can re-derive the entire replica in time kT instead of NT, and worse yet it can respond to the verifier's ℓ random challenges in time kT with only ℓ parallelism. As a result to ensure the server is storing at least $1/k$ fraction of blocks the replication time must be at least a factor N/k longer than the polling period.

Dependency graphs One way to characterize the issue with the simple cipher block chaining method is that the *dependency graph* of the block encodings is not depth robust. Let each block of the file represent a node in a graph where a directed edge is placed between the ith node and the jth node if the encoding of the jth block of the file depends on the encoding of the ith block. The resulting graph is a directed acyclic graph (DAG). By the properties of H and VDE.Enc the dependencies are cryptographically enforced: if the jth block is dependent on the ith block then the jth encoding cannot be computed unless the ith encoding is known except with negligible probability (although proving this formally is non-trivial and will take some work later). If the dependency graph is (α, β) depth robust then deleting any αN fraction of the encodings will contain a dependency path of length βN inside the deleted subgraph, meaning that it will require at least βN sequential calls to VDE.Enc to re-derive the deleted blocks. On the other hand, the dependency graph of the cipher block chained Basic-VDE-PoRep is a line, and as demonstrated by the time/space tradeoff attack described above it is at most $(1 - 1/k, k/N)$ depth robust for any $k < N$ (as storing only every kth node partitions the deleted set of nodes into lines of length k). More generally we can consider the *parallel pebbling complexity* of the dependency graph (Section 2.8).

4.3 DRG PoRep

The construction DRG-PoRep [26] extends the Basic-VDE-PoRep by chaining block dependencies using a depth robust chaining as described above. As a PoS, it can be viewed as the generic PoS from the graph labeling game (Section 2.8) on a depth robust graph, where the random oracle queries are combined with VDE queries to introduce more asymmetry between initializing the PoS and extracting data from it. Using an (N, α, β, d)-DRG we are able to reduce the time delay T for each block encoding as N increases, such that the total time NT remains the same
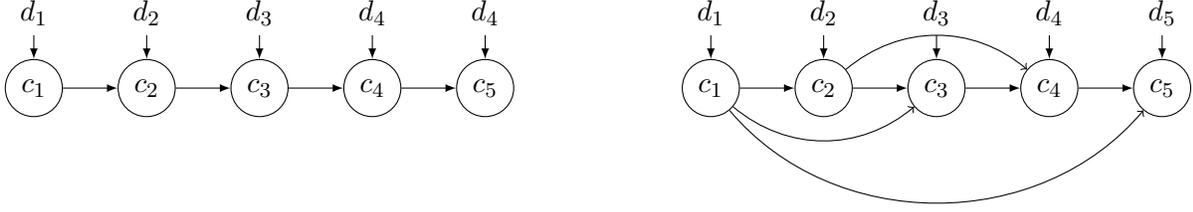
Figure 4.3: Illustration of block dependency DAG configurations in cipher block chaining encodings. On the left is a simple chain (as in the chained Basic-VDE-PoRep) whereas the right depicts a mock depth robust chaining. For each chained encoding, the ith encoding is derived as $R_i \leftarrow \mathsf{Enc}(k_i, d_i)$ where $k_i = H(id || \mathsf{parents}(i))$ and $\mathsf{parents}(i)$ denotes the set of encodings on nodes j with a directed edge to i.

and the polling period is tuned to βTN. A prover that deletes more than an α fraction of the block encodings will not be able to respond to challenges correctly (and quickly enough) during the challenge-response period. This achieves α-rational replication and replication time that is a factor $1/\beta$ longer than the polling period. The security can be amplified using stronger DRGs for smaller $\alpha > 0$, at the cost of increasing the degree by $O(1/\alpha)$ as well as the replication time relative to polling period.

As in the generic PoS from labeling games, the protocol separates out two kinds of proofs. As a part of the aux output during the replication the prover generates a proof that the depth robust chaining of the encodings were "mostly" correct. Unlike in the Basic-VDE-PoRep where the verifier can completely verify the correctness of a block using VDE.Dec, here the verifier cannot check that all the correct dependencies were enforced as this would not be a compact proof. Here, the prover will first derive the entire encoding (consisting of labels on each node of the graph) and provide the verifier with a compact vector commitment to these labels. The verifier will query for several randomly sampled nodes of the graph and challenge the prover to open the commitment to labels on both this node and the labels on all of its parent nodes. The verifier can then check that the encodings correctly observed the dependency. This convinces the verifier that a constant fraction of the nodes in the graph are correct. Because the graph is (α, β) depth robust, the (sufficiently large) subgraph of correct nodes is also depth robust. Put another way, the small fraction of incorrect labels do not help the adversary any more than putting several additional initial pebbles on the graph "for free". The proof is made non-interactive using the Fiat-Shamir heuristic.

The second "online" proof is a simple proof of retrievable commitment. The verifier simply challenges for several indices of the prover's vector commitment to the replica block encodings and the prover sends back these openings.

4.3.1 DRG-PoRep

PoRep.Setup$(\lambda, \kappa, T) \rightarrow pp$: The setup obtains as input security parameters λ, κ, as well as the delay parameter T, sets $\Delta = 2T/(\beta N - 1)$ to be the VDE delay parameter and runs $pp_{\mathsf{vde}} \leftarrow \mathsf{VDE.Setup}(\lambda, \Delta)$. This determines a block size m and $\mathcal{M} = \{0, 1\}^m$. The setup then

runs $pp_{\text{vc}} \leftarrow \text{VC.Setup}(1^\lambda, N_{max}, \mathcal{M})$ where N_{max} is the maximum supported data length[27]. Finally the setup also defines two integers $\ell_1 = \ell_1(\lambda)$ and $\ell_2 = \ell_2(\kappa)$. The setup outputs $pp = (pp_{\text{vde}}, pp_{\text{vc}}, m, \ell_1, \ell_2)$.

PoRep.Replicate$(id, \tau_D, \tilde{D}) \rightarrow R, aux$: The input to the replicate procedure is the preprocessed data file $\tilde{D}$ consisting of N blocks of size m, along with data tag τ_D and replica identifier id.

1. Apply random oracle $H(id||\tau_D) = \sigma$. Implicitly σ defines a graph $G \leftarrow \text{DRG.Sample}(N, \sigma)$, however the prover does not need to run this function explicitly as it will use the parent function instead.

2. Parse $\tilde{D}$ as data blocks $(d_1, ..., d_N)$, each $d_i \in \{0,1\}^m$. We define the following subroutine DRGEnc that operates on $\vec{d}, m, N$, and σ and outputs the replica R.

$$
\begin{aligned}
&\text{DRGEnc}(\vec{d}, m, N, \sigma)\{ \\
&\quad \textbf{for } i = 1 \ to \ N: \\
&\qquad (v_1, ..., v_d) \leftarrow \text{DRG.Parents}(N, \sigma, i) \\
&\qquad k_i \leftarrow H(\sigma||c_{v_1}|| \cdots ||c_{v_d}) \\
&\qquad c_i \leftarrow \text{VDE.Enc}(pp_{\text{vde}}, k_i \oplus d_i) \\
&\quad R \leftarrow (c_1, ..., c_n) \\
&\quad \textbf{return } R\}
\end{aligned}
$$

3. Compute vector commitment $com_R, \tau_R \leftarrow \text{VC.Com}(pp_{\text{vc}}, R)$.

4. Now use H to derive the (non-interactive) a challenge vector $\rho = (\rho_1, ..., \rho_{\ell_1})$ by computing $\rho_i = H(\sigma||com_R||i)$. Set $C^{\text{nodes}} = (c_{\rho_1}, ..., c_{\rho_{\ell_1}})$. For each i set $\text{parents}(\rho_i) \leftarrow \text{DRG.Parents}(N, \sigma, \rho_i)$ and set $C^{\text{parents}}(\rho_i) = (c_{v_1}, ..., c_{v_d})$ where $\{v_1, ..., v_d\} = \text{parents}(\rho_i)$.

5. Compute vector commitment opening proofs on the indices specified by the challenges:

$\Lambda \leftarrow \text{VC.Open}(pp_{\text{vc}}, C^{\text{nodes}}, \rho, \tau_R)$
for $i = 1$ to ℓ_1 :
$\quad \Lambda_i \leftarrow \text{VC.Open}(pp_{\text{vc}}, C^{\text{parents}}(\rho_i), \text{parents}(\rho_i), \tau_R)$

6. Output R and $\text{aux} = com_R, \sigma, \rho, C^{\text{nodes}}, C^{\text{parents}}(\rho_1), ..., C^{\text{parents}}(\rho_{\ell_1}), \Lambda, \Lambda_1, ..., \Lambda_{\ell_1}$.

PoRep.Poll$(N) \rightarrow r$: For $i = 1$ to ℓ_2 randomly sample $r_i \xleftarrow{\text{R}} [N]$. Output $r = (r_1, ..., r_{\ell_2})$.

PoRep.Prove$(R, aux, id, r) \rightarrow \pi$: The input is the replica $R = (c_1, ..., c_N)$, id, aux output of replicate (which includes com_R), a challenge vector $\vec{r} = (r_1, ..., r_{\ell_2})$. Derive $\sigma = H(id||\tau_D)$. Set $\vec{c} = (c_{r_1}, ..., c_{r_{\ell_2}})$. For each r_i also derive $\text{parents}(r_i) \leftarrow \text{DRG.Parents}(N, \sigma, r_i)$, set $C^{\text{parents}}(r_i)$ to be the concatenation of c_j for all $j \in \text{parents}(r_i)$ (ordered by increasing indices), and compute $k_{r_i} = H(\sigma||C^{\text{parents}}(r_i), i)$. Set $\vec{k} = (k_{r_1}, ..., k_{r_{\ell_2}})$.

Compute $\Lambda \leftarrow \text{VC.Open}(pp_{\text{vc}}, com_R, \vec{c}, \vec{r}, \tau_R)$. Output $\pi = (\Lambda_{\text{ret}}, \vec{c}, \vec{k})$.

---

[27]Some vector commitments, such as a Merkle commitment, do not require specifying the data length in the setup. In this case $N_{max} = \infty$.

PoRep.Verify$(id, \tau_D, r, \mathsf{aux}, \pi)$: Parse the input aux as a list of values com_R, σ, ρ, C^{nodes}, $C^{\mathsf{parents}}(\rho_1),...,C^{\mathsf{parents}}(\rho_{\ell_1})$, Λ, $\Lambda_1,...,\Lambda_{\ell_1}$ as well as the input $\pi = \Lambda_{\mathsf{ret}}, \vec{c}$. Parse $\pi = (\Lambda_{\mathsf{ret}}, \vec{c}, \vec{k})$.

1. First verify aux.[28] Check $H(id||\tau_D) = \sigma$ and $H(\sigma||com_R||i) = \rho_i$ for each $i = 1$ to ℓ_1. If any checks fail output 0 and terminate. Next derive $\mathsf{parents}(\rho_i) \leftarrow \mathsf{DRG.Parents}(N, \sigma, \rho_i)$. Run the following verifications on the vector commitment openings $\Lambda, \Lambda_1, ..., \Lambda_{\ell_1}$:

   $b_0 \leftarrow \mathsf{VC.Verify}(pp_{\mathsf{vc}}, C^{\mathsf{nodes}}, \rho, \Lambda)$
   for $i = 1$ *to* ℓ_1 :
   $\qquad b_i \leftarrow \mathsf{VC.Verify}(pp_{\mathsf{vc}}, C^{\mathsf{parents}}(\rho_i), \mathsf{parents}(\rho_i), \Lambda_i)$
   If $b_0 \wedge b_1 \wedge \cdots \wedge b_{\ell_1} = 0$ then output 0 and terminate.

   Finally for each $c_{\rho_i} \in C^{\mathsf{nodes}}$ the key $k_i = H(\sigma||c_{v_1}||\cdots||c_{v_d})$ where $\{v_1, ..., v_d\} = \mathsf{parents}(\rho_i)$ and compute $d_i \leftarrow \mathsf{VDE.Dec}(pp_{\mathsf{vde}}, k_i \oplus c_{\rho_i})$ and query the check oracle $b \leftarrow \mathcal{O}_{\mathsf{check}}(d_i, \rho_i, \tau_D)$. If $b = 0$ output 0 and terminate.

2. Second verify the "online" proof π. First compute $b \leftarrow \mathsf{VC.Verify}(pp_{\mathsf{vc}}, \vec{c}, \vec{r}, \Lambda_{\mathsf{ret}})$. Next for each $i \in [\ell_2]$ compute $d_i \leftarrow \mathsf{VDE.Dec}(pp_{\mathsf{vde}}, k_i \oplus c_{r_i})$ and query the oracle $b'_i \leftarrow \mathcal{O}_{\mathsf{check}}(d_i, r_i, \tau_D)$. Output $b \wedge b'_1 \wedge \cdots \wedge b'_{\ell_2}$.

Replication and extraction time   The main achievement of the DRG PoRep construction is its scalable replication time, which is now a fixed multiple of the polling period (dependent only on α, β) and in particular is independent of the number of blocks in the file. The extraction time is also optimal, as any block can be locally decoded by applying the hash function H to $d = O(\log n)$ labels and making a single call to $\mathsf{VDE.Dec}$, which should take under 1 second.

Batched vector commitment openings   In the construction description above we listed the vector commitment openings Λ, $\Lambda_1,...,\Lambda_{\ell_1}$ for C^{nodes} and all $C^{\mathsf{parents}}(\rho_{\ell_i})$ separately inside the parameter aux, and they were verified separately as well. This was done mostly for clarify of presentation. This would be necessary if the vector commitments are instantiated with Merkle commitments. However, one of the major advantages of using vector commitments with batchable openings is that these can all be compressed to a single constant size opening. In this case aux only includes the labels C^{nodes} and all $C^{\mathsf{parents}}(\rho_{\ell_i})$ for each i along with a single opening proof Λ.

Vector commitment storage   If the prover uses a Merkle tree as its vector commitment then it will either need to additionally store the hashes on internal Merkle nodes or recompute them on the fly. At first glance this may appear to destroy the tightness of the PoS because storing the Merkle tree is certainly not a PoS. However, because the time/space tradeoff in storing this tree is so smooth the honest prover can delete the hashes on nodes on the first k levels of the tree to save a factor 2^k space and re-derive all hashes along a Merkle path by reading at most 2^k nodes and computing at most 2^k hashes. If $k = 7$ this is less than a 1% overhead in space, and requires at most 128 additional hashes and reads. Furthermore, as remarked in [45] these 2^k

---

[28]In the interactive protocol, as long as the verifier is stateful then the input aux only needs to be verified the first time the verifier runs **PoRep.Verify** on its first poll as it can remember the verification result for subsequent polls.

reads are sequential memory reads, which in practice are inexpensive compared to the random reads for challenge labels.

Relaxing soundness: interaction and time/space tradeoffs   The size of ℓ_1 is a significant factor in blowing up the size of aux. The reason why in general we need to set ℓ_1 to be large is that otherwise the prover can brute force a favorable challenge in the non-interactive proof. There are two ways to improve on this. First, we could get rid of the non-interactive proof in aux and required the verifier to challenge the prover for a fresh aux proof on their first interaction. Second, the non-interactive challenge "grinding" attack ignores the fact that the prover will need to rerun replication many times in expectation. As this is already a memory/sequentially hard computation (taking at least around 10-60 min), we can relax the security model to assume there is some bound on the initial computation that the malicious prover is willing to do in order to save a fraction of space during the online phase. If we can just achieve soundness $1/1000$ instead of 2^{-80} then even a massively parallelized prover will need to grind in expectation for 3.47 days. For this soundness level we can set $\ell_1 \approx 100$.

Proof sizes   There are two types of proofs, the "offline" non-interactive proof contained in aux and the "online" proofs output by PoRep.Prove. The proof contained in aux is much larger (although it is only sent once to each verifier, which amortizes the cost). The proof size in aux is $\ell_1(d+1)m + L(N, m, \ell_1(d+1))$ where $L(N, m, \ell_1(d+1))$ is the size of the vector commitment opening proof on a vector of length N of elements of size m. As aux is non-interactive the soundness error needs to be exponentially small, therefore we set $\ell_1 = \lambda/-log(1-\alpha)$. For example with $\lambda = 80$ and $\alpha = 0.20$ this is $\ell_1 = 825$. However, as described above, in practice we could set $\ell_1 = 100$ if we are willing to assume a malicious prover will not find it rational to work for 4 days to save a fraction of the space. With $d = 5$ and $m = 128$ the total proof size is approximately 10 KB, independent of the data input size. Note that the data input size could range up to gigabytes or terabytes.

Compressing proofs with SNARGs   We can further compress the size of the aux proof by using SNARGs, or any other form of succinct proofs. To maximally compress the proof size, the prover can compute a SNARG over the circuit PoRep.Verify with the proof π as a witness. To optimize the performance of this SNARG it would be important to choose instantiations of vector commitments and slow encodings that minimize the multiplicative complexity of checking inclusion proofs and decoding. If the main cost in the proof size comes from the degree d of the DRGs (see Section 5), then we can already substantially decrease the proof size by using SNARGs only to eliminate the parent nodes of each c_i. In this case, the SNARG would only prove knowledge of d encodings $c_1, ..., c_d$, their valid inclusion proofs in com_R at the appropriate positions, and that $k_i = H(c_1, ..., c_d)$. We can also replace the hash function H with a suitable *key-derivation function* (KDF) that minimizes multiplicative complexity. Since the proof of security holds in the random oracle model, it is important that the KDF should retain properties similar to a random oracle. At a minimum it needs to be collision-resistant.

The Jubjub[29] Pedersen hash over BLS12-381 is an attractive candidate as it achieves a circuit complexity of only 4 multiplication gates per input bit (61,440 gates per Merkle commitment opening with a 32GB file). This can be used to instantiate both Merkle commitments and the

---

[29]https://z.cash/technology/jubjub.html

function H. If the VDE is instantiated with Sloth++ [14] over the scalar field of BLS12-381, then verification of a 5 min delay involves roughly 6 million multiplication gates. This is due to the fact that Sloth++ iterates a square-root round function 30 million times. However, with a 1GB file the delay T will be reduced drastically, e.g. with $\beta = 1/100$ it will require only 3 iterations of the Sloth++ round function. Furthermore, if a longer delay (on the order of 5 minutes) is necessary then we can instead use a VDE based on inverting permutation polynomials over $\mathbb{F}_p$ described in [14], which is a nominal added cost in combination with the Merkle inclusion proof. For sequential security this would require tuning the polling period to the time it would take a prover running on an industry standard GPU as the polynomial GCD computations do admit some parallelism. The total circuit size for verifying aux with $m = 256$, $N = 2^{30}$, $d = 20$, and $\ell_1 = 100$ is approximately 124 million gates, and would take approximately 5 hours to compute on a single-threaded 2.4 GHz Intel E7-8870 CPU [12]. With modest parallelism (e.g. 16 threads) this can easily be reduced to below the raw replication time.

Towards both tight and efficient PoReps   The DRG-PoRep construction improved significantly on replication time while maintaining terrific extraction efficiency. However, it compromised on ϵ-rational security and the tightness of the proof of space. In order to achieve ϵ-arbitrarily small, we required DRG graphs that were robust in ϵ-fraction subgraphs. Not only does this degrade the practicality of the DRG construction, it also worsens the gap between polling and replication time, which necessarily increases as $O(1/\epsilon)$. Thus it does not really achieve arbitrarily small ϵ-rational security in the same sense as Basic-PoRep. Furthermore, by increasing dependencies between blocks, erasure code preprocessing is no longer guaranteed to recover the file from arbitrary deletion of an ϵ fraction of the blocks (only the ϵN topologically last blocks).

Pietrzak [45] also proposed using depth robust graphs for proofs of replication and presented a slightly different construction to DRG-PoRep that partially resolves the issues of space-tightness in an elegant way. The construction uses the recent [6] DRGs which are $(n, \alpha, \beta, O(\log n/\epsilon))$-depth robust for all (α, β) such that $1 - \alpha + \beta \geq 1 - \epsilon$. Instead of embedding the data on all nodes of the graph, the construction generates a graph on $4n$ nodes, but only encodes the data on the topologically last n nodes. The replication procedure still generates a labelling of the entire graph upon which the last n block encodings are dependent, but only outputs the labels on the last n nodes. Pietrzak shows that a prover who deletes an ϵ' fraction of the labels on the last n nodes will not be able to re-derive them in fewer than n sequential steps. The value ϵ' can be made arbitrarily small however $\epsilon < \epsilon'/4$, so the degree of the graph must increase as $1/\epsilon'$. Furthermore, although the graphs in [6] achieve asymptotic efficiency and are incredibly intriguing from a theoretical perspective, they still do not have concretely practical degrees (their asymptotic degree starts to kick in on graphs with over a billion nodes).

Pietrzak's construction did not incorporate delay encodings into the DRG PoRep construction and this could easily be added just as in DRG-PoRep. If only SHA256 is called for each labeling this means that only graphs of a minimum size over 1 billion nodes can achieve a 10 min delay. SHA256 can be intentionally slowed with iterations, but then extraction becomes as inefficient as the replica generation. Hence the main advantage of the VDEs is in lowering the gap between extraction/replication on graphs of smaller size. However, since Pietrzak's construction only embeds the data on the last level the data extraction is already as inefficient as data replication (the labels on the first $3n$ labels needs to be re-derived before the data can be extracted), hence the VDEs will not help as much.

5 Instantiating Depth Robust Graphs

DRG-PoRep requires an (n, ϵ, δ, d) DRG where $\epsilon, \delta < 1$ and $d \in \text{polylog}(n)$. In general for performance we want to minimize ϵ and d while maximizing δ. Recall that the value of ϵ determines the ϵ-rational-security achievable, and space gap of the construction as proof of space. The degree impacts the PoRep proof size and verification complexity, which is $O(\lambda d)$.

Explicit Depth Robust Graphs Erdős et. al. [43] showed how to construct DRGs explicitly from extreme constant-degree bipartite expander graphs. Using their construction, one can obtain an $(n, \alpha, \beta, c \log n)$ DRG on n nodes for particular constants, e.g. $\alpha = 0.99$ and $\beta = 0.1$, and sufficiently large n. The constant factor c depends on the degree of the bipartite expander graphs used in the iterated construction. While explicit constructions of these graphs exist [41], they are complex and have either large constant degree or only achieve the requisite expansion properties for a significantly large number of nodes. Mahmoody et. al. [36] use denser bipartite expander graphs to construct for any $\epsilon < 1$ a DRG family that is $(n, \alpha, \alpha - \epsilon, c \log^2 n)$ depth robust for all $\alpha < 1$. Again, instantiating this construction with explicit expanders will result in graphs that have an impractically large degree. There is a new construction by Alwen et. al. [6] that improves asymptotically on MMV, but not concretely.

Probabilistic Constructions If we compromise on explicit constructions then we can instead use probabilistic methods to sample much simpler graphs with more practical concrete parameters that satisfy the desired depth robustness property with high probability. Intuitively, since random graphs have good expansion properties in expectation, we can replace the explicit expanders used inside the DRG constructions with randomly sampled graphs instead. Alwen et. al. [5] proposed and analyzed a more direct probabilistic DRG sampling algorithm that outputs a $(n, 1 - \alpha/\log n, \beta, 2)$ DRG on n nodes with failure probability negligible in n. Their construction can be easily modified to output a $(n', 1 - \alpha, \beta, c \log n')$ DRG on $n' = O(n/\log n)$ nodes for some fixed constant c. Unfortunately, their analysis only shows that the output graph is a DRG with good probability for very high values of $1 - \alpha$. On the other hand they provide strong empirical evidence that the output graph retains depth robustness for much smaller subgraphs than analyzed theoretically, and thus provides hope that a tighter analysis would yield much better values of α (and even β).

5.1 Bucket Sampling

The starting point of our DRG sampling algorithm is the algorithm of Alwen et. al. [5], which produces a degree-2. The algorithm is extraordinarily simple. It is a small tweak on a random DAG, sampling d edges for each node at index v randomly from the preceding nodes at indices $u < v$, but biasing the selection towards nodes that are closer to v.

The construction operates on nodes with integer indices in $[n]$. First, a directed edge connects each node u to its successor $u + 1$. Next, let $B_i = \{(u, v) : 2^{i-1} \leq |v - u|\}$ where $u, v \in [n]$ are the integer indices of nodes u and v respectively. For a given node v, let $B_i(v)$ denote the set of all $u' < v$ such that $(u', v) \in B_i$, i.e. it contains the nodes u that are within a "distance" in the range $[2^{i-1}, 2^i]$ from v. For each node v, a bucket B_i with $i \leq log_2 v$ is selected uniformly at random and then finally a random node $u \xleftarrow{\text{R}} B_i$ is selected. The edge (u, v) is added to the graph.

```
BucketSample[n]{
    V := [n];  E := ∅
    for v = 2, ..., n :
        E := E ∪ {(v − 1, v)}
        i ⇐ᴿ {1, ..., log₂ v};  u ⇐ᴿ Bᵢ(v)
        E := E ∪ {(u, v)}
    return(V, E)}
```

```
BucketSample[n, m]{
    (V, E) ← BucketSample[nm]
    W := [n];  E' := ∅
    for(i, j) ∈ E :
        uᵢ ← i % n;  uⱼ ← j % n
        E' := E' ∪ {(uᵢ, uⱼ)}
    return(W, E')}
```

This construction, denoted BucketSample[n], outputs a graph that is asymptotically *block depth robust* with overwhelming probability. Furthermore, if the sampling is fixed by a seed σ to a pseudorandom generator then BucketSample[n] is both deterministic and locally navigatable. The function DRG.Parents(n, σ, v) is naturally defined by the construction, which would use the seed σ to sample the parents of node v. The definition of block depth robustness is as follows.

Definition 24 (ABH17). *A (n, α, β, m, d) block depth robust graph is directed acyclic graph G with d-regular indegree on n nodes indexed by integers in $[n]$ such that if any set S of size $(1-\alpha)n$ and its left neighborhood set $D_m(S) = \{x \in G : \exists u \in S \text{ s.t. } 0 \le u - x \le m\}$ are removed, then the graph $G \setminus D_m(S)$ contains a path of length at least βn.*

The bucket sampling algorithm of Alwen et. al. [5] outputs a $(n, 1 - c_1 / \log n, \beta, c_2 \log n, 2)$ block depth robust graph with failure probability negligible in n. Note that (n, α, β, m, d) block depth robustness is only possible for $m(1-\alpha) < 1-\beta$, so the bucket sampling algorithm outputs a graph whose block depth robustness is within a constant factor (i.e., $(1 - \beta)/(c_1 c_2)$) of the optimal. According to the concrete lower bounds on parameters proved in their analysis, for $\beta = 0.03$ they get $m(1-\alpha) > 160 \cdot 2.43 \times 10^{-4} > 0.038$, which is within a factor 25.5 of optimal. Next, we show how to use the block depth robust BucketSample[n] construction to construct larger degree DAGs with better depth robustness. The construction BucketSample[n, m] outputs a graph on n nodes of degree m that is a "factor" m more depth robust, i.e. improves $(n, 1-\alpha, \beta, 2)$ depth robustness to $(n, 1 - \alpha m, \beta, m + 1)$ depth robustness.

"Metagraph" construction   Suppose we are given an algorithm that for any (sufficiently large) n outputs a graph that is $(n, 1 - \alpha, \beta, m, d)$ block depth robust. We construct a new graph G on nodes in $[n]$ of degree dm as follows. First construct a graph G' on nodes in $[nm]$. We define the graph G such that there is a directed edge from node i to node j if and only if G' contains a directed edge (u, v) for some $u \in [(i - 1)m + 1, i \cdot m]$ and $v \in [(j - 1)m + 1, j \cdot m]$. It is easy to see that if G' has in-degree d then the graph G has in-degree at most dm. Following the terminology of Alwen et. al., we call G the *metagraph* of G', which can also notate as $G = G'_m$. Using the underlying DRG construction BucketSample[n] the corresponding metagraph construction BucketSample[n, m] actually has degree at most $m + 1$ because one of the two edges is to the immediate predecessor.

We prove the following lemma, which essentially says that G inherits the absolute depth robustness of G' except now on a vertex set of $1/m$ the size.

Lemma 5. *If G is an $(mn, 1 - \alpha, \beta, m, d)$ block depth robust robust graph then its meta graph G_m is $(n, 1 - \alpha m, \beta, dm)$ depth robust.*

Proof. Denote by B_i the interval $[(i-1)m+1, i \cdot m]$ of nodes in G. Let $\mathcal{B} = \{B_1, ..., B_n\}$. As noted in the graph construction, each node of the meta graph G_m corresponds to one of the intervals $B_i \in \mathcal{B}$, and has degree at most dm. Consider removing at most $e = \alpha mn$ nodes from G_m. This corresponds to removing at most e of the intervals in $\mathcal{B}$ from G. By the block depth robustness of G, since $e = \alpha mn$ then the remaining set of nodes in G contains a path of length βmn. Any path of length βmn nodes in G must intersect at least $\lceil \beta n \rceil \geq \beta n$ intervals in $\mathcal{B}$ because a path intersecting at most $k < \beta n$ intervals of length m would contain at most $km < \beta mn$ nodes. This implies that there remains a path of length at least βn in the meta graph G_m. $\qquad\square$
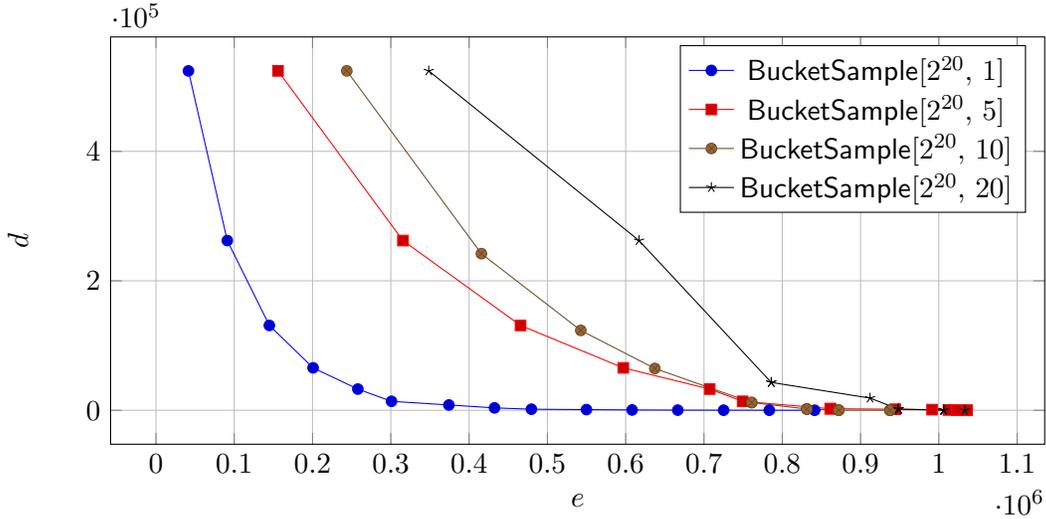
Figure 5.1: Results of the attacks against BucketSample$[n, m]$ for m = 1, 5, 10 and $n = 2^{20}$. We plot on the y-axis for each value of $e < n$ on the x-axis the smallest depth among any of the subgraphs of size $n - e$ that any of the attacks were able to find. For example, with BucketSample$[n, 5]$ the attacks could not locate any subgraph on 70% of the nodes that had depth below $n/4$. With BucketSample$[n, 20]$ the best attack on 10% subgraphs reduced the depth to $n/64$.

Analytical and empirical results Looking under the hood in the analysis of [5], the meta-graph G_m with $m = 160 \log n$ is analytically a $(n, 0.961, 0.3, 160 \log n)$ depth robust graph with overwhelming probability. Alwen et. al. also give an empirical analysis on their 2-regular graph construction for $n = 2^{24}$ nodes where they implement the best known depth reducing attacks to locate subsets of various sizes that contain short paths. The graph in their paper shows pairs (d, e) where d is the lowest depth found in any subgraph of size at least $n - e$. For example, their experiment results show that for $e = 0.2 \times 10^7$ the smallest depth found was around $4 \times 10^6 \approx 0.24n$ nodes, suggesting that the sampled graph is $(2^{24}, 0.88, 0.24, 2)$ depth robust. We implemented the same attacks against the larger degree graphs output by BucketSample$[n, m]$, which are mostly based on a classical algorithm by Valiant [50] for locating a depth reducing set, shown below in Figure 5.1. The empirical results suggest that the graph output by BucketSample$[n, 5]$ on $n = 2^{20}$ is $(n, 0.70, 1/4, 6)$ depth robust, that BucketSample$[n, 10]$ is

$(n, 0.28, 1/32, 11)$ depth robust, and that $\mathsf{BucketSample}[n, 20]$ is $(n, 0.10, 1/64, 21)$ depth robust, retaining high depth even in 10% subgraphs.

6 Formal security analysis

This section applies the framework and definitions of Section 3 to formally analyze the security of the two constructions $\mathsf{Basic\text{-}VDE\text{-}PoRep}$ and $\mathsf{DRG\text{-}PoRep}$. The focus of our analysis is on proving ϵ-rational-security. In fact, Lemma 4 established sufficiency conditions that greatly simplify proving ϵ-rational security. Both constructions are each a δ-PoRC of the replica R and data D almost by construction (and PRAM strongly sound as proven in Lemma 1), and the main work left is show that each is a parallel sound proof of space. Both of the constructions can be viewed as PoS variants of the generic graph labeling PoS (with a tweak for incorporating VDEs). Furthermore, due to Pietrzak's recent result [45], we can contain our analysis to parallel pebbling games, which translates to unconditional security in the random oracle model with some small loss (Theorem 2). (Before this result graph labeling PoS constructions were still analyzed via pebbling games and the correspondence was taken for granted). There is a small catch in relying on this correspondence in our setting, however, which is that Pietrzak's analysis applied to pure random oracle labeling games and did not involve VDEs.

6.1 Labeling games with VDEs

Pebbling game model Pebbling games appropriately model labeling games with collision resistant one-way hash functions informally because of the property that labels on distinct nodes do not repeat with any significant probability and computing new labels cannot be predicted without knowing all the inputs to the labeling function, i.e. all the dependency labels. The independence of labels even holds when there are no dependency edges by incorporating the unique index of the node as the first input. In the labeling games with VDEs that we consider, a collision resistant hash function (or random oracle) is used to first derive a key from all the dependencies of a node. The key is then XORed with a (non-unique) data input, and the result is passed to a VDE to derive the node's label. Conditioned on fixed data inputs, these "key labels" are uniquely determined, and are derived as the output of collision resistant hash functions just as in usual labeling games.

The important distinction, however, is that the call to the VDE is much more expensive than the call to the hash function to derive the key. The cost of "placing a pebble" is dominated by the VDE time delay. The sequentiality security property of a VDE (Definition 13) says that evaluating the VDE on a point will be hard on a *random* input, even given an arbitrary polynomially sized table of evaluations on other points. If the hash function used to derive the key labels is modeled as random oracle then the input to each VDE call is an independent uniform random point.

As we are focussed on a parallel adversary, we may assume for the purpose of our analysis that the adversary always calls the VDE to derive the label whenever a key and input is available. In fact, for the purpose of analyzing PoS security of PoRep constructions we may assume that the data inputs are (they are assumed to be available externally in the worst case). This way the underlying pebbling game is largely unchanged, and we can model the VDE and random oracle queries as a single pebbling move that can be made only when there are labels on all the dependencies.

Committed data inputs As we already noted, it is crucial that the auxiliary data inputs are committed (or publicly known) and that the labeling is uniquely dependent on this commitment. Otherwise, the prover can place any arbitrary labels on the nodes and use the VDE decode algorithm to find corresponding data inputs consistent with these labels and the key derived from labels on all neighboring nodes. This allows the prover to circumvent the labeling dependencies of the underlying graph or even the cost of evaluating the VDE. The issue is resolved by using the data commitment τ_D of all data inputs to sample the random oracle H_σ. Intuitively, the pebbling game is still an accurate security model because the adversary must "know" D before labeling the graph using H_σ.

Parallel pebbling vs labeling Intuitively, any labeling strategy starting from an initial advice string that simply stores a set of graph labels then this directly translates to a pebbling strategy in the corresponding pebbling game and thus inherits the security of the pebbling game. Moreover, when the labels themselves encode entries in a random oracle's function table they are incompressible (Fact 2). However, the labeling adversary's initial advice (equiv. the PoS adversary's persistent storage) need not encode these labels explicitly.

Pietrzak's unconditional proof of this equivalence in the random oracle model first demonstrates how to translate a certain set F called "fresh" queries (roughly those queries that contain any label which has not yet been received as an output of another query) made by a successful labeling strategy into a successful parallel pebbling strategy that uses $|F|$ initial pebbles. Hence, if the pebbling game is hard without $\Omega(n)$ initial pebbles then $|F| = \Omega(n)$. Intuitively, the labeling adverary's initial advice s must somehow encode F, and therefore should have at least the same size as F is incompressible (Fact 2). The next step of the proof demonstrates this formally by showing how to construct an encoding scheme that uses the adversary as a black box to encode these $|F|$ entries in the random oracle's function table using just $\approx |s|$ bits.

It is not immediately clear that this equivalence still holds when comparing the parallel pebbling game with the VDE cost function to the parallel hardness of the VDE labeling game. Pietrzak proved the equivalence still holds when the auxiliary data inputs are simply XORed with the random oracle "key" outputs, and assuming the commitment τ_D is extractable (e.g., a commitment that uses the random oracle itself, such as a Merkle commitment, is extractable in the random oracle model). In this case the encoding scheme in the proof works roughly the same way but first extracts the data inputs from the adversary and uses the data to recover the incompressible "keys". In the constructions we analyze in this paper the "keys" are similarly XORed with the data inputs before they are passed to the VDE. Moreover, the VDE is invertible so the the key labels are still extractable from the output given the data.

However, the security of the construction relies on the cost of querying the VDE, which is significantly greater than querying the random oracle. One way to address this is to use the IDP oracle (Definition 14) to model queries to the VDE separately from queries to the random oracle, and then adapt the analysis of [45] to show that an adversary who wins the labeling storing too little advice either can compress the random permutation table of VDE.Enc or the random function table of H_σ. This analysis is beyond the scope of the current document.

6.2 Security claims

For the following claims we assume that $\mathcal{O}_{\text{check}}$ is perfectly binding on pairs (τ_D, D) such that each τ_D is a position binding vector commitment to D, i.e. for each τ_D and i there exists a

unique D_i such that $\mathcal{O}_{\text{check}}(\tau_D, i, D, \text{aux}) = 1$ for some aux.

Claim 8. *Basic-VDE-PoRep is (PRAM strongly) μ-sound $(1-\delta)$-PoRC for any $\mu > (1-\delta)^{\ell(\lambda)} +$ negl(λ) and with constant additive extractor blowup.*

Proof. The implicit PoRC protocol of Basic-VDE-PoRep is just an instantiation of PoRC from vector commitments Section 2.4.2. The vector commitment is (τ_D, N) and the opening advice is (R, id). Each $R_i = \text{VDE.Enc}(d_i \oplus H(id||\tau_D||i))$ where H is a collision hash function and each output of VDE.Enc is uniquely decodable. VC.Open on τ_D, an index $i \in [N]$, block d_i, and the opening advice R_i computes $\text{VDE.Dec}(R_i) \oplus H(id||\tau_D||i) = d_i$ and then verifies $\mathcal{O}_{\text{check}}(\tau_D, i, D, \text{aux}) = 1$. This satisfies position binding by the assumption that $\mathcal{O}_{\text{check}}$ is position binding. Moreover, since R_i is uniquely determined from d_i by the properties of VDE and collision resistance of H the commitment to d_i is also position binding to R_i. In the PoRC protocol each challenge in the challenge vector specifies an index i for the ith pair (R_i, d_i), and the proof value R_i is an opening of this pair. Therefore the claim follows from Theorem 1. $\square$

Claim 9. *The composition of k instances of Basic-VDE-PoRep on distinct ids $id_1, ..., id_k$ is a parallel $((1 - \epsilon + \delta)kN, 2T, (1 - \epsilon + \delta)^{\ell(\lambda)})$-sound PoS for any $1 > \epsilon > \delta > 0$.*

Proof. The implicit PoS in a single instance of Basic-VDE-PoRep is a graph labeling PoS on a disconnected graph on N, with $V_C = [N]$, and Chal the uniform distribution over $[N]$. The cost of placing a single pebble is T, i.e. $c(N) = \Delta = 2T$, i.e. when the VDE delay Δ is set to $2T$. The random black pebbling game on a disconnected graph on N nodes with $V_C = [N]$ is $(\gamma, 1, \gamma)$-parallel-hard for any γ because the challenge node is unpebbled with probability at least $1 - \gamma$ and it requires 1 move/round to pebble an unpebbled challenge node. Since there are no edge dependencies the adversary cannot place any red pebbles, i.e. $\delta = 0$. Therefore, Claim 4 can actually be strengthened in this case to say the PoS is parallel $(\gamma N, T, \gamma^k)$-sound. Composition follows from Claim 5. Setting $\gamma = 1 - \epsilon + \delta$ gives $\mu = (1 - \epsilon + \delta)^k$. $\square$

Theorem 3. *Basic-VDE-PoRep with VDE delay parameter $\Delta = 2T$ is $e^{-\lambda}$-sound ϵ-rational secure when $\ell(\lambda) = 2\lambda/\epsilon$.*

Proof. Claim 8 proves condition (b) of Lemma 4 with extractor blowup factor $c \approx 1$ and Claim 9 proves condition (c) of Lemma 4. Together with the correctness of the construction for an honest prover, these claims therefore establish that Basic-VDE-PoRep is μ-sound ϵ-rational secure for any $\mu > max((1-\delta)^\ell, (1 - \epsilon + \delta^\ell))$ and any $\delta < \epsilon$. The maximum value of μ is at $\delta = \epsilon/2$, where $\mu = (1 - \epsilon/2)^\ell$. If $\ell = \frac{2\lambda}{\epsilon}$ then $(1 - \frac{\epsilon}{2})^\ell = (1 - \frac{\epsilon}{2})^{2\lambda/\epsilon} < e^{-\lambda}$. $\square$

Claim 10. *DRG-PoRep is a (PRAM strongly) μ-sound $(1-\delta)$-PoRC for any $\mu > (1-\delta)^{\ell_2(\kappa)} +$ negl(λ) with constant additive PRAM extractor blowup.*

Proof. We similarly claim that the implicit PoRC protocol of DRG-PoRep is a PoRC from vector commitments with the security parameter $\ell_2(\kappa)$ determining the number of challenges. Here the vector commitment explicitly uses a vector commitment scheme VC as an abstract primitive. The commitment is (τ_D, com_R, N) where com_R is a vector commitment to the replica R consisting of labels on the DRG nodes. The opening advice for an index i is the label $R_i = c_i$ on the ith node along with the key k_i derived from the parent labels. The opening value d_i is then computed as $\text{VDE.Dec}(c_i \oplus k_i) = d_i$ and the opening to d_i is then checked by querying the oracle on $\mathsf{O}_{\text{check}}(\tau_d, i, D, \text{aux})$. Additionally the opening value R_i is verified against com_R using the VC

functionality. Correctness is immediate and binding follows from the assumption that $\mathcal{O}_{\mathsf{check}}$ is position binding and the security properties of VC (i.e. it is also position binding). The opening advice is contained in the "online proof" Λ_{ret}. Note that PoRep.Verify also runs a verification procedure on aux (this is important for the PoS property), however this is not necessary for the PoRC verification and can be ignored for the purpose of showing that this is a PoRC. Finally the claim follows from Theorem 1. $\qquad\square$

Claim 11. *The composition of k instances of DRG-PoRep on distinct ids $id_1, ..., id_k$ is a parallel $((1 - \epsilon + \delta)kN, 2T, \mu)$-sound PoS for $\mu = max\{(1 - \delta)^{\ell_1(\lambda)}, (1 - \epsilon + \alpha + 2\delta)^{\ell_2(\lambda)}\}$ for any $\delta > 0$ when the graph used is an (N, α, β) DRG and the delay parameter is $\Delta = 2T/(\beta N - 1)$.*

Proof. The implicit PoS protocol of each instance of DRG-PoRep is the generic graph labeling PoS on a (N, α, β) depth robust graph. By Lemma 2, the graph labeling PoS on a family $G(N)$ of (N, α, β) depth robust graphs with labeling cost function $c(N)$ is an $(\gamma n, (\beta N - 1)c(N), max\{(1 - \delta)^{\ell_1}, (\alpha + \gamma + \epsilon)^{\ell_2}\}$-parallel-sound PoS for any $\delta < 1$. Setting $\gamma = 1 - \epsilon + \delta$ gives $\mu = max((1 - \delta)^{\ell_1(\lambda)}, (1 - \epsilon + 2\delta + \alpha)^{\ell_2(\kappa)})$. For simplicity we set $\lambda = \kappa$ (recall the distinction was to heuristically allow for different levels of security for online vs. offline proofs). Finally, the pebbling cost is $c(N) = 2T$ when the VDE delay parameter $\Delta = 2T/(\beta N - 1)$. Moreover, this PoS is secure under composition (Lemma 2). $\qquad\square$

Theorem 4. *DRG-PoRep on an (N, α, β) DRG and with VDE delay parameter $\Delta = 2T/(\beta N - 1)$ is $e^{-\lambda}$-sound ϵ-rational secure when $\ell_1 < \ell_2$ and $\ell_1(\lambda) = \lambda/\delta$ and $\ell_2(\lambda) = \lambda/(\epsilon - \alpha - 2\delta)$ for any $0 < \delta < (\epsilon - \alpha)/2$.*

Proof. Claim 8 proves condition (b) of Lemma 4 with extractor blowup factor $c \approx 1$ and Claim 9 proves condition (c) of Lemma 4. Together with the correctness of the construction for an honest prover, these claims therefore establish that DRG-PoRep is μ-sound ϵ-rational secure for any:

$$\mu > max\{(1 - \delta)^{\ell_2}, (1 - \delta)^{\ell_1}, (1 - \epsilon + \alpha + 2\delta)^{\ell_2})$$

In general for practicality it is desirable that ℓ_1 is as small as possible, whereas ℓ_2 may be larger. Therefore, we may assume that $\ell_1 < \ell_2$ and can ignore the $(1 - \delta)^{\ell_2}$ term. Finally, using the inequality $(1 - x)^x < e^{-x}$ for $x \in (0, 1)$ derives yields the theorem statmeent with $\ell_1 = \lambda/\delta$ and $\ell_2 = \lambda/(\epsilon - \alpha - 2\delta)$. $\qquad\square$

The parameter δ in Theorem 4 allows one to tune the relative size of ℓ_1 and ℓ_2 to maintain the same level of security in the parameter λ. Decreasing δ increases ℓ_1 and decreases ℓ_2.

Acknowledgements

References

[1] Filecoin: A decentralized storage network. Protocol Labs, 2017. `https://filecoin.io/filecoin.pdf`.

[2] Proof of replication. Protocol Labs, 2017. `https://filecoin.io/proof-of-replication.pdf`.

[3] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In *ASIACRYPT*, 2017.

[4] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, pages 191–219, 2016.

[5] Joël Alwen, Jeremiah Blocki, and Benjamin Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In *CCS*, 2017.

[6] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In *EUROCRYPT*, 2018.

[7] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication. In *25th USENIX Security Symposium*, 2016.

[8] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security*, 2007.

[9] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *Asiacrypt*, 2009.

[10] Niko Baríc and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 480–494, 1997.

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[13] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *EUROCRYPT*, 1993.

[14] Dan Boneh, Joseph Bonneau, Benedikt Bunz, and Ben Fisch. Verifiable delay functions. 2018. To appear in CRYPTO 2018.

[15] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: theory and implementation. In *CCSW'09 Proceedings of the 2009 ACM workshop on Cloud computing security)*, 2009.

[16] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, 2002.

[17] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *PKC 2013*, 2013.

[18] F.R.K. Chung. On concentrators, superconcentrators, generalizers, and nonblocking networks. In *Bell System Technical Journal*, 1979.

[19] Bram Cohen. Proofs of space and time. Blockchain Protocol Analysis and Security Engi-

neering, 2017. `https://cyber.stanford.edu/sites/default/files/bramcohen.pdf`.

[20] Moni Naor Cynthia Dwork and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO*, 2005.

[21] Ivan Damgard, Chaya Ganesh, and Claudio Orlandi. Proofs of replicated storage without timing assumptions. Cryptology ePrint Archive, Report 2018/654, 2018. `https://eprint.iacr.org/2018/654`.

[22] Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and prgs. In *CRYPTO*, 2010.

[23] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference (TCC)*, 2009.

[24] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO*, 2015.

[25] Uriel Fiege, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. In *Journal of Cryptology*, 1988.

[26] Ben Fisch, Joseph Bonneau, Juan Benet, and Nicola Greco. Proofs of replication using depth robust graphs. In *Blockchain Protocol Analysis and Security Engineering 2018*, 2018. `https://cyber.stanford.edu/bpase2018`.

[27] Ben Fisch and Shashwat Silas. Weak compression and (in)security of rational proofs of storage. Cryptology ePrint Archive, Report 2018/514, 2018. `https://eprint.iacr.org/2018/514`.

[28] Philippe Golle, Stanislaw Jarecki, and Ilya Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography*, 2002.

[29] Chethan Kamath Vladimir Kolmogorov Krzysztof Pietrzak Joöel Alwen, Binyi Chen and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In *EUROCRYPT*, 2016.

[30] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.

[31] Arjen K Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptology ePrint Archive*, 2015, 2015.

[32] Hendrik W Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[33] Sergio Demian Lerner. Proof of unique blockchain storage, 2014. `https://bitslog.wordpress.com/2014/11/03/proof-of-local-blockchain-storage/`.

[34] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional commitment schemes: From polynomial commitments to pairing-based from simple assumptions. In *ICALP*, 2016.

[35] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC*, 2010.

[36] Mohammad Mahmoody, Tal Moran, and Salil P Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO*. Springer, 2011.

[37] Ueli Maurer. Unifying proofs of knowledge. In *CRYPTO*, 2009.

[38] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, 2004.

[39] S. Matthew Weinberg Miles Carlsten, Harry Kalodner and Arvind Narayanan. On the instability of bitcoin without the block reward. In *ACM CCS*, 2016.

[40] Tal Moran and Ilan Orlov. Rational Proofs of Space-Time. Cryptology ePrint Archive #

2016/035, 2016.

[41] Salil Vadhan Omer Reingold and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In *FOCS*, 2000.

[42] Sunoo Park, Krzysztof Pietrzak, Albert Kwon, Joël Alwen, Georg Fuchsbauer, and Peter Gai. Spacemint: A cryptocurrency based on proofs of space. Cryptology ePrint Archive, Report 2015/528, 2015. `http://eprint.iacr.org/2015/528`.

[43] Ronald L. Graham Paul Erdös and Endre Szemeredi. On sparse graphs with dense long paths. In *Computers & Mathematics with Applications*, 1975.

[44] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

[45] Krzysztof Pietrzak. Proofs of Catalytic Space. Cryptology ePrint Archive # 2018/194, 2018.

[46] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In *TCC*, 2016.

[47] Randal Burns Reza Curtmola, Osama Khan and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *In Distributed Computing Systems, 2008. ICDCS08.*, 2008.

[48] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Asiacrypt*, 2008.

[49] Luca Trevisan, Salil Vadhan, and David Zuckerman. Compression of samplable sources. *Computational Complexity*, 14(3):186–227, 2005.

[50] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *Mathematical Foundations of Computer Science*, 1977.

[51] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *ACM CCS*, 2012.

[52] Gaven J. Watson, Reihaneh Safavi-Naini, Mohsen Alimomeni, Michael E. Locasto, and Shivaramakrishnan Narayan. Lost: location based storage. In *CCSW*, 2012.