Scaling Proof-of-Replication for Filecoin Mining

Ben Fisch[1], Joseph Bonneau[2], Nicola Greco[3], and Juan Benet[3]

[1]Stanford University
[2]New York University
[3]Protocol Labs

Abstract

A *proof-of-replication* (PoRep) is a proof system that a server can use to demonstrate to a network in a publicly verifiable way that it is dedicating unique resources to storing one or more replicas of a data file. While it is not possible for PoReps to guarantee cryptographically that the prover's storage format is redundant, PoReps do guarantee that:

(a) The prover must be using as much space to produce the proof as replicas it claims to store (it is a proof of space)

(b) The prover can retrieve a committed data file (it is a proof of retrievability)

(c) The prover can use the space to store this file without any overhead

In this sense a PoRep is a *useful proof of space.* It is uniquely suited to replace proof-of-work in Nakamoto consensus as a Sybil resistance mechanism, while simultaneously incentivizing and subsidizing the cost of file storage.

Technical report This is a short technical report on our constructions. A more detailed paper is forthcoming with information about our prototype implementation of PoReps.

1 Proofs of Replication

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. Parallel algorithms operate in the PRAM model.

1. $\mathsf{PoRep.Setup}(\lambda, T) \rightarrow pp$ 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.

2. $\mathsf{PoRep.Preproc}(sk, D) \rightarrow \tilde{D}, \tau_D$ 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. $\mathsf{PoRep.Replicate}(id, \tau_D, \tilde{D}) \rightarrow R, \mathsf{aux}$ 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. $\mathsf{PoRep.Extract}(pp, id, R) \rightarrow \tilde{D}$ on input replica R and identifier id outputs the data $\tilde{D}$.

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

6. $\mathsf{PoRep.Poll}(\mathsf{aux}) \rightarrow r$: This takes as input the auxiliary replica information aux and outputs a public challenge r.

7. $\mathsf{PoRep.Verify}(id, \tau_D, r, \mathsf{aux}, \pi) \rightarrow \{0, 1\}$ on input replica identifier id, data tag τ_D, public challenge 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. The setup (whether a deterministic, trusted, or transparent public setup) 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 $(\tilde{D}, \tau_D) \leftarrow \mathsf{PoRep.Preproc}(sk, D)$. 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 $\mathsf{PoRep.Setup}$ is run by a trusted party[1] 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 may use a secret-key, but it is not trusted (in particular it may collude with the prover). A 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 previous notions of proof of data replication [6, 27].

ϵ-Rational Replication An ideal security goal for PoRep protocols would be to guarantee the following properties, described informally:

> Any prover who simultaneously passes verification in k distinct PoRep protocols (under k distinct identities) where the input to $\mathsf{PoRep.Replicate}$ 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.

By "storing k independent replicas" we mean that τ is a *k-replication* of a file D if the string τ can be partitioned into k substrings $\tau_1, ..., \tau_k$ such that each allow full recovery of the file D. More generally, we say that τ is a k-replication of a vector of (possibly repeating data blocks)

[1]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

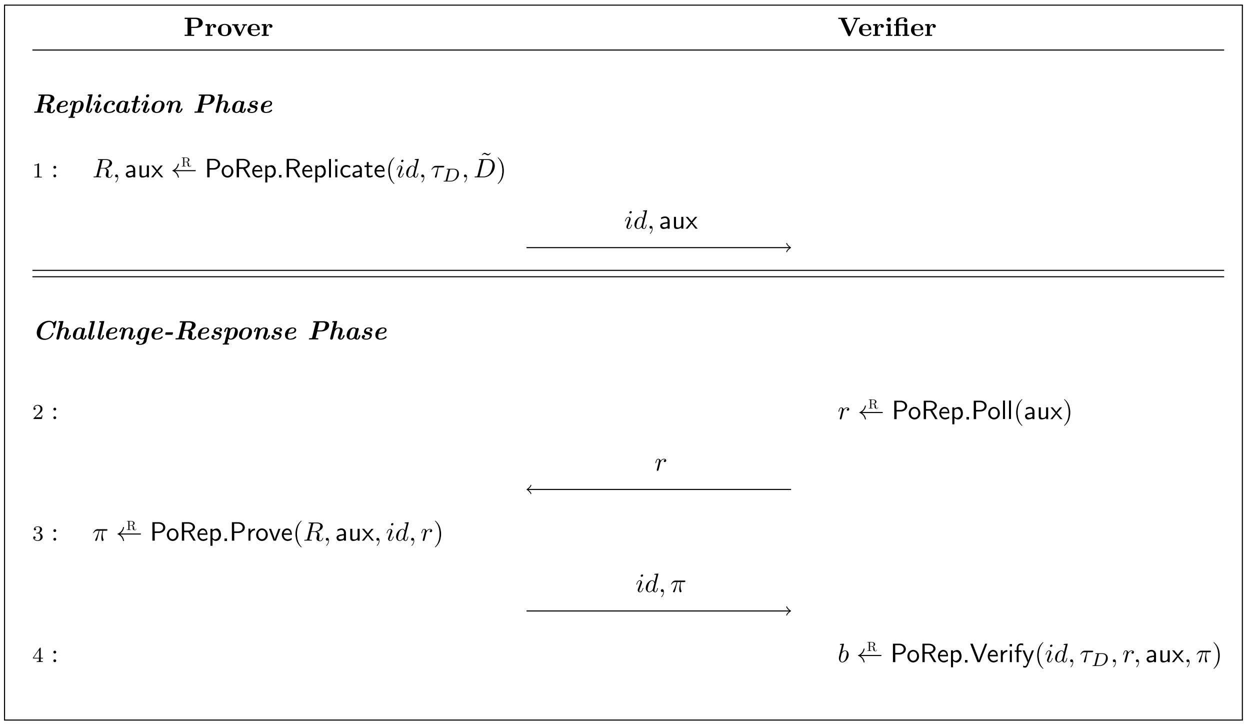
Prover
Verifier
Replication Phase
1 : $R, \mathsf{aux} \xleftarrow{R} \mathsf{PoRep.Replicate}(id, \tau_D, \tilde{D})$
id, aux
Challenge-Response Phase
2 : $r \xleftarrow{R} \mathsf{PoRep.Poll}(\mathsf{aux})$
r
3 : $\pi \xleftarrow{R} \mathsf{PoRep.Prove}(R, \mathsf{aux}, id, r)$
id, π
4 : $b \xleftarrow{R} \mathsf{PoRep.Verify}(id, \tau_D, r, \mathsf{aux}, \pi)$

Figure 1.1: 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.

$D_1, ..., D_k$ if τ can be partitioned into k substrings as above such that for each D_i there is a unique τ_i from which D_i can be fully recovered. (Note how this is the similar to the definition of an optimal erasure code with rate $1/k$, yet a weaker requirement as the data in an erasure code can be recovered from any $1/k$ fraction of the bits).

This would imply that if several provers each provide distinct PoReps of the same file then they are each dedicating unique resources[2] 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[3]), as we explain next.

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.

[2]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.

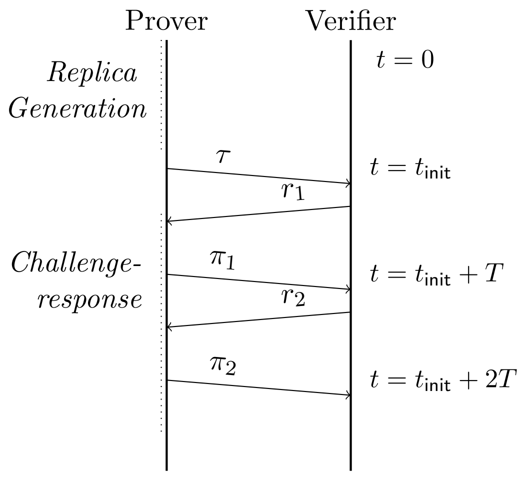

Figure 1.2: Space-time diagram of the PoRep protocol. Following a phase of length t_{init} during which the prover generates a new replica, the verifier repeatedly challenges the prover to produce a PoRep within a challenge time period length T in order to verify that the prover is still storing the unique replica of the original data. For this proof system to be sound it is necessary that $t_{\mathsf{init}} >> T$.

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.

Instead, we will need to relax the security model to consider adversaries who are "honest-but-opportunistic", meaning they will only employ a malicious strategy if they stand to save more than some ϵ cost doing so (measured in storage resources). This security model, called ϵ-rational replication, has been formally specified and applied rigorously to analyze the constructions included in this report [13]. In the context of the Filecoin storage network and blockchain ecosystem, ϵ-rational replication captures the ϵ cost that clients must pay to convince miners to encode their real data inside PoReps rather than "useless" generated data, and therefore the degree to which Filecoin subsidizes storage costs.

Proof of space A PoRep is a publicly verifiable proof-of-space (PoS) [12]. A prover that passes verification in the interactive challenge-reponse protocol for a file $\tilde{D}$ of claimed size $|\tilde{D}| = N$ must be using $\Omega(N)$ persistent storage. Moreover, a prover that passes verification in k instances of this protocol with distinct ids $id_1, ..., id_k$ and claimed file sizes $N_1, ..., N_k$ must be using $\Omega(M)$ space where $M = \sum_{i=1}^{k} N_i$. This is implied by ϵ-rational replication and is in general a weaker property.

Data preprocessing and data retrievability Finally, a PoRep is a proof-of-retrievability (PoR) [16] of the underlying data represented by the data tag τ_D. The type of security guarantee here depends on the mode of the data preprocessing step. When the data input is preprocessed using a secret-key the resulting PoRep is a public-coin PoR. In this scenario we can imagine

[3]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 [30]

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 (i.e. anyone with the tag τ_D can verify on behalf of the client). When the preprocessor runs in keyless mode the resulting PoRep is a publicly-verifiable *proof of retrievable commitment* (PoRC)[4]. In this case τ_D is simply a binding commitment to the data file D, and the PoRep is a proof that the prover can retrieve 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 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.

2 Basic PoRep from Sequential Encodings

The first basic PoRep we describe applies a *slow encoding* to a file F to transform it into a format $\tilde{F}$, which can be quickly decoded back to F. This general approach has been described before in [1,9,19]. The slow transformation is done in an initialization period "offline". A verifier then periodically checks that the server is still storing the encoding $\tilde{F}$. If the server has deleted the encoding $\tilde{F}$ then it will not be able to re-derive it quickly enough to respond to challenges from the verifier during the "online" phase. Furthermore, the encoding can be made unique to a particular identifier *id*, so that two the encodings $\tilde{F}_{id1}$ and $\tilde{F}_{id2}$ (called *replicas*) are independent and cannot be quickly derived from one another. This way, a server that is only storing one of the two encodings of the same original file F will fail the online challenges from the verifier for the missing replica.

Verifiable Delay Encodings The primitive we use to implement slow encodings in our most basic PoRep is called a *verifiable delay encoding* (VDE). These will also play a role in our more advanced constructions. Informally, as described, this is an encoding that is slow to compute yet fast to decode. More specifically, 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 VDF [9]. Practical examples of VDEs include Sloth [17], MiMC [3], and a special class of permutation polynomials [9].

The syntax we will use for a VDE is a tuple of three algorithms $\mathsf{VDE} = \{\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}\}$ defined as follows. (Some constructions of $\mathsf{VDESetup}$ may require this to be run by a trusted party, or computed using MPC. The instantiations described above do not).

1. $\mathsf{VDE.Setup}(t, \lambda) \to pp$ is given security parameter λ and delay parameter t produce public parameters $\mathbf{pp}$. 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.

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

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

[4]This primitive is formally defined in [13]. It is similar to a proof of knowledge of a commitment, only with a public extraction property more similar to PoR. The publicly verifiable Merkle commitment described in [16] is a simple example of a PoRC

2.1 Basic PoRep Construction

In all of the constructions we describe in this report we will skip the description of $\mathsf{PoRep.Preproc}$. 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 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 $\mathcal{O}_{\mathsf{check}}(d_i, i, \tau_D) \rightarrow b \in \{0,1\}$. The construction will use a VDE scheme $\{\mathsf{VDE.Setup}, \mathsf{VDE.Enc}, \mathsf{VDE.Dec}\}$ with identical input space and code space over $\{0,1\}^m$, as well as a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^m$ modeled as a random oracle (i.e. maps strings of arbitrary length to strings of length m). For two strings $s_1, s_2 \in \{0,1\}^*$ the notation $s_1||s_2$ denotes their concatenation.

PoRep.Setup$(\lambda, T) \rightarrow pp$ Run $\mathsf{VDE.Setup}(T, \lambda) \rightarrow pp$. This specifies the block length m, and provides implicit input parameters to $\mathsf{VDE.Enc}$ and $\mathsf{VDE.Dec}$.

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

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

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

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

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

Instantiation We can instantiate the basic PoRep construction with the Sloth [18] VDE. 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 ($N = 100$) take under 1 hour and extraction takes under 1 second on 16 parallel cores. With block size $m = 256$ we can only support files to 320 bytes for replication to take 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.

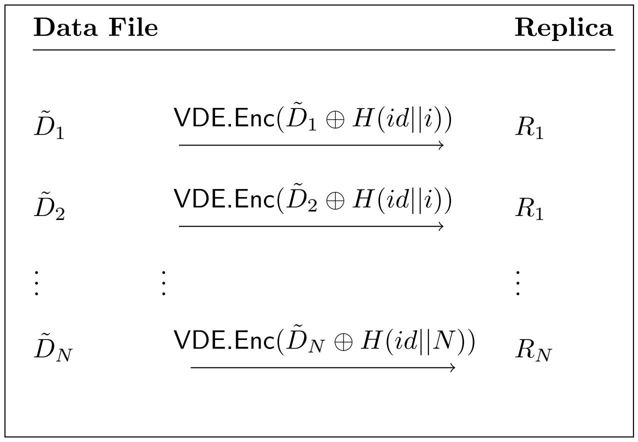
Data File
Replica
$\tilde{D}_1$
$\mathsf{VDE.Enc}(\tilde{D}_1 \oplus H(id||i))$
R_1
$\tilde{D}_2$
$\mathsf{VDE.Enc}(\tilde{D}_2 \oplus H(id||i))$
R_1
$\vdots$
$\vdots$
$\vdots$
$\tilde{D}_N$
$\mathsf{VDE.Enc}(\tilde{D}_N \oplus H(id||N))$
R_N

Figure 2.1: Illustration of PoRep.Replicate in the basic PoRep construction using a VDE.

Proof size The proof size is λm bits. To detect deletion of 1% of the data block with soundness error 1/3 we would require $\lambda = 100$, in which case the proof might as well include the entire replica (if we support only up to $N = 100$). To detect deletion of 5% with soundness 1/3 only requires $\lambda = 20$, or 1/5 of the entire file. For 80% we can set $\lambda = 5$, or 5% of the file size. In combination with erasure code during preprocessing, the file may still be perfectly recoverable. Therefore we still achieve ϵ-replication. For example, if D is preprocessed with an erasure code that tolerates arbitrary 20% deletion, then we tradeoff an increase in the replica size by 20% for a proof size of only 6% of the original file size.

3 Block Chaining Sequential Encodings

The Basic-VDE-PoRep does not scale well 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 time TN.

This is not just due to paranoia about massively parallel attacks! Any attacker who uses only a factor k more parallelism will be able to reduce replication time by a factor k. If the best adversary can generate replicas a factor k faster then the time to replicate for the honest provers must be at least a factor k times longer than the polling period. In fact, since the verification strategy randomly samples only ℓ blocks to challenge, the adversary only needs ℓ parallelism to re-derive the challenged blocks in wall-clock time T.

Block chaining 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-

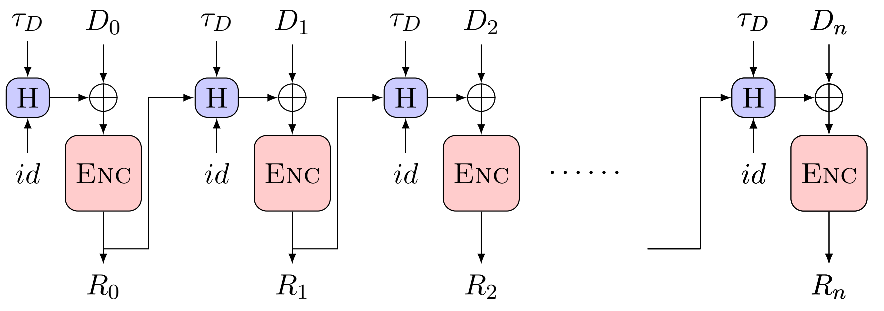

Figure 3.1: Basic-VDE-PoRep in CBC-mode.

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 3.1.

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. However, 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.

Depth robust graphs What is a depth robust graph? An (n, α, β, d) depth robust graph is a DAG on n nodes with in-degree d such that any αn size subgraph contains a path of length βn.

Definition 1. *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 $\sigma \xleftarrow{R} \{0,1\}^s$ 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 $G_\sigma \leftarrow DRG.Sample(n, \sigma)$.*

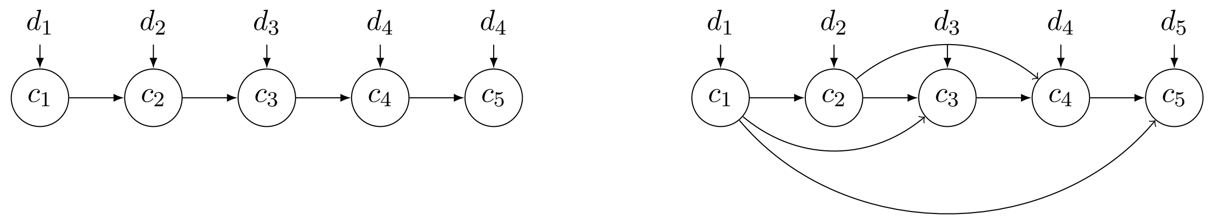

Figure 3.2: Illustration of block dependency DAG configurations in cipher block chaining encodings. On the left is a simple chain (as in the chained Basic-VDE-PoRep) whereas the right depicts a mock depth robust chaining. For each chained encoding, the ith encoding is derived as $R_i \leftarrow \mathsf{Enc}(k_i, d_i)$ 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.

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 $1 - \mathsf{negl}(n)$ over $\sigma \overset{\text{R}}{\leftarrow} \{0,1\}^s$.

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).

Pebbling complexity More generally we can consider the *pebbling complexity* of the dependency graph. This is defined in terms of a game where the player is allowed to place an initial number of pebbles on nodes of the graph. The game then proceeds in steps where in each step the player is allowed to place a new pebble on the graph with the restriction that it can only place a pebble on a node if all of its parents currently have pebbles. The player may also remove pebbles at any point. The game ends when the players has pebbled all nodes in some target set. There are various measures of pebbling hardness that have to do with the minimum number of steps required to pebble the target set from an initial configuration of a given size. The *parallel pebbling complexity* of the graph is captured by a similar game with the modification that in any "round" of the game the player can simultaneously pebble any node whose parent nodes had pebbles in the previous round, and is hard if from an initial configuration of some maximum size the adversary requires a minimum number of rounds to pebble the target set. Finally, a random pebbling game is one where the challenge node is selected randomly and hardness is measured in terms of the probability a player can win this game from a configuration of some size and some maximum number of moves/rounds.

Proofs of space Many proofs of space [12, 25, 26] and memory hard functions [14, 15] are based on iterated collision-resistant hash function computations with hard-to-pebble dependency graphs, called a *labeling* of the graph. Depth robust graphs were also used for publicly verifiable proofs of sequential work [21]. The generic approach to constructing a PoS from a pebbling-hard graph proceeds in two steps: in an "offline" phase the prover commits to the labeling and demonstrates that the labeling is mostly correct (i.e. edge dependencies were respected) by answering random challenges to the commitment, and in an "online" phase the prover demonstrates that it can retrieve most of the labeling. Our PoRep constructions also follow this structure. Combining the labeling game with sequential encodings results in a smooth spectrum bridging the two techniques. On one end of the spectrum, large files with very large block dependency graphs will not need a large time-delay and are therefore nearly equivalent to proofs of space with data XORed into the labels. On the other end of the spectrum, very small graphs will not benefit from chaining and are therefore nearly equivalent to the basic VDE PoRep.

3.1 Depth Robust Chaining of Sequential Encodings

Our new PoRep DRG-PoRep extends the Basic-VDE-PoRep by chaining block dependencies using a depth robust chaining as described above. 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 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. Unfortunately, erasure codes no longer guarantee that the data is still recoverable from an α fraction of the encodings due to the dense block dependencies. However, 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.

Online and offline proofs The protocol separates 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. The verifier cannot check that all the correct dependencies were enforced as this would not be a compact proof. Instead, the prover derives the entire encoding (consisting of labels on each node of the graph) and provides the verifier with a compact vector commitment to these labels. The verifier queries for several randomly sampled nodes of the graph and challenges 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 depth robust, a sufficiently large subgraph of correct nodes is also depth robust. Actually, we will make this proof 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.

3.1.1 DRG-PoRep Construction

PoRep.Replicate$(id, \tau_D, \tilde{D}) \to R, aux$

1. $H(id||\tau_D) = \sigma$. This is the seed used to "sample" the DRG graph and "salt" the hash function.

2. Parse $\tilde{D}$ as data blocks $(d_1, ..., d_N)$. Run DRGEnc on $\vec{d}, m, N$, and σ. The output is the replica R.

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

3. Compute a Merkle commitment[5] com_R to the replica blocks R.

4. Now use H to non-interactively derive the challenge vector $\rho = (\rho_1, ..., \rho_{\ell_1})$ as $\rho_i = H(\sigma || com_R || i)$. These specify a set of challenge nodes $C^{\mathsf{nodes}} = (c_{\rho_1}, ..., c_{\rho_{\ell_1}})$. The proof will provide the labels on the challenge nodes, demonstrate that they were committed to in com_R, and that they are at least locally consistent with their parent labels.

   For each i set $\mathsf{parents}(\rho_i) \leftarrow \mathsf{DRG.Parents}(N, \sigma, \rho_i)$ and set $C^{\mathsf{parents}}(\rho_i) = (c_{v_1}, ..., c_{v_d})$ where $\{v_1, ..., v_d\} = \mathsf{parents}(\rho_i)$.

5. Compute Merkle proofs Λ_{nodes} that all the challenge node labels C^{nodes} and Merkle proofs $\Lambda_{\mathsf{parents}}$ that their parent labels C^{parents} are all consistent with the commitment com_R.

6. Output R and $\mathsf{aux} = com_R, C^{\mathsf{nodes}}, \Lambda_{\mathsf{nodes}}, C^{\mathsf{parents}}, \Lambda_{\mathsf{parents}}$.

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

PoRep.Prove$(R, aux, id, r) \rightarrow \pi$**:** For each r_i in the challenge vector $\vec{r} = (r_1, ..., r_{\ell_2})$ derive the key for the node r_i as $k_{r_i} = H(\sigma || C^{\mathsf{parents}}(r_i), i)$ where $R = (c_1, ..., c_n)$. Provide Merkle inclusion proof Λ_i for each c_{r_i}, and set $\Lambda_{\mathsf{ret}} = \Lambda_1, ..., \Lambda_\ell$, set $\vec{c} = (c_{r_1}, ..., c_{r_\ell})$, and set $\vec{k} = (k_{r_1}, ..., k_{r_\ell})$. Output the proof containing the Merkle proofs and key/label pairs, $\pi = (\Lambda_{\mathsf{ret}}, \vec{c}, \vec{k})$.

PoRep.Verify$(id, \tau_D, r, aux, \pi) \rightarrow b$ 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, Λ_1,...,Λ_{ℓ_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.[6] 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 reject the proof. Verify the Merkle proof Λ_{nodes} on C^{nodes}. Next, for each i derive $\mathsf{parents}(\rho_i) \leftarrow \mathsf{DRG.Parents}(N, \sigma, \rho_i)$ and the key $k_{\rho_i} = H(\sigma || C^{\mathsf{parents}}(\rho_i), i)$. Check that $d_i = \mathsf{VDE.Dec}(pp_{\mathsf{vde}}, k_i \oplus c_{\rho_i})$. 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 verify the Merkle proof Λ_{ret} for the challenge vector $\vec{c}$. 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 $\mathcal{O}_{\mathsf{check}}(d_i, r_i, \tau_D)$ If any Merkle proof verifications or oracle queries fail reject the proof.

[5] More generally, Merkle commitments can be replaced with any vector commitment [10, 20].

[6] 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.

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. There are two security parameters, ℓ_1 is a security parameter determining the number of queries in the offline proof, and ℓ_2 is a security parameter determining the number of queries in the online proof. The proof size is roughly $(d\ell_1 + \ell_2) \times m \times \log N$ bits because the responses to the ℓ_1 challenge queries include the d parents of each challenge node whereas the responses to the ℓ_2 queries do not. Vector commitments with constant size or batched openings, as opposed to Merkle commitments, would reduce the factor $\log N$ overhead. As aux is non-interactive the soundness error needs to be exponentially small, therefore we set $\ell_1 = \lambda / -\log(1-\alpha)$. For example to achieve 80-bit security and $\alpha = 0.20$ this is $\ell_1 = 825$.

Relaxing soundness: interaction and time/space tradeoffs 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 requires the prover to rerun replication many times in expectation. As this is already a memory/sequentially hard computation (taking at least around 10-60 min), we could incorporate this into the rational security model. The question is how much extra replication work a 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 $\ell_1 \approx 100$.

Compressing proofs with SNARGs We can further compress the size of the aux proof by using SNARGs, or any other form of succinct proofs. The prover computes 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, slow encodings, and collision resistant hash functions that minimize the multiplicative complexity of checking inclusion proofs and decoding. If the VDE is expensive to implement in a SNARG circuit, then just eliminating the labels on parent nodes already substantially decreases the proof size.

The Jubjub[7] 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 function H. If the VDE is instantiated with Sloth++ [9] 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 [9]. 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 [8]. With

[7]`https://z.cash/technology/jubjub.html`

modest parallelism (e.g. 16 threads) this can easily be reduced to below the replication time.

3.2 Stacked DRG PoRep

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, in some sense we cheated, as the Basic-PoRep achieves arbitrarily small ϵ-rational security for a fixed replication time. In our final construction we show how to overcome this at the expense of a slower extraction time. Our basic idea is to layer DRGs, iterating the DRG-PoRep construction so that each layer "re-encodes" the previous layer. In our first variant, we add additional edge dependencies *between* the layers that are used in the encoding. These edge dependencies are the edges of a bipartite expander graph between the layers. A bipartite expander graph has the property that every constant size subset of sinks is connected to a relatively large subset of sources. Unfortunately, by adding these additional edge dependencies between the layers they can no longer be decoded. Therefore, in this variant we instead use the first $\ell - 1$ layers to deterministically derive the keys that are used to encode the data input on the last layer. Data extraction is as expensive as data replication because it requires re-deriving the keys in the same way.

As we will show, this has the effect of amplifying the DRG security by exponentially blowing up the dependencies between nodes on the last level and nodes on the first level. Deletion of a small ϵ fraction of node labels on the last level will require re-derivation of nearly all the node labels on the first level (or even first several levels). Therefore, a relatively weak (constant) depth-robustness on the first few levels is all that is necessary to guarantee parallel pebbling hardness. In particular, we are able to prove that an $(n, 0.80n, \Omega(n))$ DRG is sufficient, i.e. deletion of 20% of nodes leaves a high depth graph on the 80% remaining nodes, regardless of the value of ϵ. Moreover, the number of levels required to achieve this is only $O(\log(1/\epsilon))$. This results in a factor $\log(1/\epsilon)$ gap between polling and replication time as opposed to the previous $O(1/\epsilon)$ gap.

Tight PoS In fact, this construction also gives the first concretely practical (and provably secure) *tight* proof of space, outperforming [2, 12, 25, 26] in its ability to achieve arbitrarily tight proofs of space with fixed degree graphs. The only other tight PoS is Pietrzak's DRG construction [25] and requires special DRGs of degree $O(1/\epsilon)$ where ϵ is the space gap. A PoS based on pebbling a graph of degree $O(1/\epsilon)$ results in a total proof size of $O(1/\epsilon^2)$. Already by stacking $O(\log(1/\epsilon))$ fixed degree DRGs we are able to achieve proof complexity $O(1/\epsilon \cdot \log(1/\epsilon))$. However, we are able to go even further and show that the total number of queries over all layers can be kept at $O(1/\delta)$, achieving proof complexity $O(1/\epsilon)$. This is in fact the optimal proof complexity for the (generic) pebbling-based PoS with at most an ϵ space gap. If the prover claims to be storing n pebbles and the proof queries less than $1/\epsilon$ then a random deletion of an ϵ fraction of these pebbles evades detection with probability at least $(1-\epsilon)^{1/\epsilon} \approx 1/e$. The same applies if a random ϵ fraction of the pebbles the prover claims to be storing are red (i.e. errors).

Data extraction and zig-zag As we mentioned, the downside of our first variant is the data extraction time, which is now as expensive as data replication. Our second variant fixes this with a simple tweak. Instead of adding edge dependencies between the layers, we add the edges of a constant degree expander graph in each layer so that every layer is both depth-robust and has high "expansion". A non-bipartite expander graph has the property that the boundary of every constant size subset, i.e. the number of neighboring nodes, is relatively large. Technically, the graph we construct in each layer is an undirected expander, meaning that the union of the dependencies and targets of any subset in our DAG is large. However, by alternating the direction of the edges between layers, forming a "zig-zag", we are able to show that the dependencies between layers expand. Now the only edges between layers are between nodes at the same index, and the label on each node encodes the label on the node at the same index in the previous level. The dependencies used for keys are all contained in the same layer; thus, the labels in any layer can be used to recover the labels in the preceding layer. Furthermore, the decoding step can be done in parallel just as in DRG-PoRep.

It is easy to see that without alternative the direction of the edges between layers this construction would fail to be a tight proof of space because the topologically last ϵn nodes in a layer would only depend on the topologically last ϵn nodes in the previous layer. Moreover, if the prover stores the labels on the topologically first $(1-\epsilon)n$ nodes it can quickly recover the labels on the topologically first $(1-\epsilon)n$ nodes in the preceding level, allowing it to recover the missing ϵn labels as well in parallel-time $O(\epsilon n)$. This is no more secure than DRG-PoRep and cannot tolerate arbitrarily small ϵ for a fixed polling period.

Related techniques Pietrzak [25] 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 [5] 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. This is similar in spirit to our idea of layering DRGs as it divides the graph into 4 layers by depth, hower here a single DRG is defined over all the 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 [5] achieve asymptotic efficiency and are incredibly intriguing from a theoretical perspective, they still do not have concretely practical degree. (As a side point, Pietrzak's construction does not incorporate delay encodings into the DRG PoRep construction. 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).

Ren and Devadas [26] construct a proof of space from stacked bipartite expander graphs. Our construction can be viewed in some sense as a merger of this technique with DRGs, however it requires a very new and more involved analysis. Their construction can easily be modified into a "proof of replication" as well by encoding data only on the last level (similar to Pietrzak's DRG construction and our first variant of stacked DRGs). Specifically, the labels on all levels $V_1, ..., V_{r-1}$ are first derived and then each label ℓ_i on the ith node of the last level L_r is replaced with $\mathsf{VDE.Enc}(\ell_i \oplus d_i)$ where d_i is the ith block of data. The data extraction is as inefficient as replica generation because it must be extracted by re-running the computation from the start to derive the labels ℓ_i and then decoding each block. However, their construction would also not satisfy our stronger definition for PoRep security (with a time bounded adversary) as it is not secure against parallel attacks. Furthermore, the space gap is quite weak (i.e. it cannot beat $\epsilon = 1/2$ according to their analysis). Our construction is a strict improvement as it is both space-tight and secure against parallel attacks.

CBC layering Finally, we draw a comparison to a PoRep proposal hinted at in [1] which suggested iterating CBC encryption over the data with a permutation interlaced between layers. This is the direct inspiration for our construction, and it is instructive to observe the pitfalls of implementing this approach with CBC-mode encryption and why they are resolved when using a depth robust chaining mode instead.

The CBC-layering method is as follows. Let Π denote a random permutation on the block indices of the file, i.e. $\Pi : [N] \to [N]$. (In practice Π can be represented compactly using a Feistel cipher and a random seed to generate the Feistel round keys). Let $\mathsf{CBCEnc}(id, D)$ denote the $\mathsf{Basic\text{-}VDE\text{-}PoRep}$ in CBC-mode as illustrated in Figure 3.1. Additionally for any length N input vector $\vec{x}$ define the function $\mathsf{Shuffle}^{\Pi}(x_1, ..., x_N) = (x_{\Pi(1)}, ..., x_{\Pi(N)})$. Starting with the plaintext blocks $D = d_1, ..., d_N$, compute the encoding $c_1, ..., c_N \leftarrow \mathsf{CBCEnc}(id, D)$ and set $V_0 = (c_1, ..., c_N)$. Next shuffle and re-encode the blocks of V_0 so that $V_1 = \mathsf{CBCEnc}(id, \mathsf{Shuffle}^{\Pi}(V_0))$. Continue iterating the process so that $V_i = \mathsf{CBCEnc}(id, \mathsf{Shuffle}^{\Pi}(V_{i-1}))$. Output the last layer V_ℓ as the replica encoding R. Extraction can be run in the reverse direction, with a factor k parallel speedup on k parallel processors.

There are two main issues with this approach:

1. It is hard to verify that the prover maintains the edge dependencies (i.e. computes the CBC encoding correctly). If the prover cheats on only a $1/\sqrt{N}$ fraction of edges in each level then it cuts the sequential work by a factor $\sqrt{N}$ and the probability of catching the prover with only a small number of queries is very small. Basically this is for the same reason that a hash chain is not a publicly verifiable proof of sequential work[8] (unless we use SNARGs over a circuit of size $N\ell$, which would be impractical for large N).

2. The prover could use the time/space tradeoff attack on CBC-mode encodings to storing only ϵ/ℓ fraction of the block labels on each level, for total space storage ϵN and it can re-derive any block label on the last level in ℓ^2/ϵ sequential steps. For fixed polling period of 5 min, in order to catch an adversary using ϵN storage ℓ must be sufficiently large so that ℓ^2/ϵ steps (i.e. calls to $\mathsf{VDE.Enc}$) must take at least 5 min. As each call therefore takes $5\epsilon/\ell^2$ min it implies that the total replication time takes $5\epsilon N/\ell$ minutes. Let $\epsilon = 1/2$, $N = 2^{30}$ (16GB file), and consider two cases. If $N/\ell > 2^{11}$ then replication takes 3.5 days. On the other hand if $\ell > N/2^{11}$ then $\ell N = N^2/2^{11} = 2^{49}$. Even if each call to $\mathsf{VDE.Enc}$ on a 16 byte block is reduce to 1 cycle on a 2 GHz machine, 2^{49} cycles still takes 3 days.[9]

[8]In fact, MMV11 [22] use depth robust graphs for the very purpose of obtaining publicly verifiable proofs of sequential work.

[9]The attack is much worse when the permutation is not applied between layers. For any ℓ, if the adversary stores $\epsilon N/\ell$ evenly spaced columns of block labels for total space ϵN then it is easy to see that the adversary can (with unlimited parallelism) recompute all the labels on the last level in $2\ell/\epsilon$) parallel steps by deriving blocks along diagonals (lower left to upper right), deriving a block in the next level as soon as its dependencies in the

Depth robust layering Using a depth robust chaining rather than CBC on each level gets rid of both of these issues. It would be nice to prove security just from permutations between layers, however our analysis relies heavily on the additional expander edges instead of permutations. Intuitively, the permutations between layers would prevent the attacker from finding a depth reducing set of nodes whose dependencies in all previous layers also belong to a depth reducing set.

Expander construction We use the randomized construction of bipartite expander graphs originally due to Chung [11]. This samples the edges of the graph using a random permutation. More precisely, the edges of a d-regular bipartite expander on $2n$ vertices are defined by connecting the dn outgoing edges of the sources to the dn incoming edges of the sinks via a random permutation $\Pi : [d] \times [n] \to [d] \times [n]$. That is, the ith source is connected to the jth sink if there is some $k_1, k_2 \in [d]$ such that $\Pi(k_1, i) = (k_2, j)$. This has been shown to result in a bipartite expander with overwhelming probability in n [7, 26, 28]. Our construction uses 8-regular bipartite graphs constructed in this way. We also use this to construct an undirected non-bipartite expander graph by treating the N nodes as both the targets and the sinks, defining the edge (i, j) if and only if there exists $k_1, k_2 \in [8]$ such that either $\Pi(k_1, i) = (k_2, j)$ or $\Pi(k_1, j) = (k_2, i)$. We will revisit expander graphs in more depth in our formal analysis section.

Number of layers and degree In the protocol description we set the number of layers to a parameter $L = 10$. Based on our analysis, using a degree $d = 8$ expander graph and targeting $\epsilon = 1\%$ then we can safely set $L = 10$. There are two options for targeting smaller ϵ. One is to increase the degree d, but the other is to increase the number of layers. In general, for fixed expander degree $d = 8$ the number of required layers increases as $O(\log(1/\epsilon))$ as $\epsilon \to 0$. This is the better option in our opinion. Unless the file is so large that the time for each $\mathsf{VDE.Enc}$ call is optimally small (e.g. a single call to AES, or 20 cycles on an Intel CPU with AES-NI), as we increase the layers we can still maintain the same initialization time by lowering the delay parameter of each block encoding. Importantly, the difference between replication time and the polling period remains the same. We are also able to achieve constant proof complexity as we increase the number of layers. The number of queries to the vector commitment of the final layer is $O(1/\epsilon)$, but we can reduce this exponentially (i.e. by a multiplicative factor) between layers. Increasing the degree on the other hand increases the proof size.

3.2.1 Stacked-DRG-PoRep

Our presentation of the construction follows the second "zig-zag" variant.

As before, H is a random oracle $H : \{0,1\}^* \to \{0,1\}^m$ and $\Pi : \{0,1\}^\lambda \times [8] \times [N] \to [8] \times [N]$ is a random permutation oracle where for each seed $\sigma \xleftarrow{R} \{0,1\}^\lambda$ the function $\Pi(\sigma, \cdot)$ is computationally close to a random permutation of $[8] \times [N]$, where $N = \text{poly}(\lambda)$. As $N = \text{poly}(\lambda)$, Π can be realized using a cryptographic PRG to generate $\Theta(N \log N)$ pseudorandom bits from the seed σ in order to sample a random permutation, however in practice it is better if both Π and Π^{-1} can be evaluated more efficiently, i.e. in polylog(N) time and space.

previous level are available. Therefore, to catch an adversary that is using only ϵN storage, it is necessary to set the polling period $T \leq 2\ell/\epsilon$, which implies that the total replica generation time is $\ell N \geq \epsilon T N/2$, i.e. a factor $\epsilon N/2$ longer than the polling period. If the polling period is 5 minutes, $\epsilon = 1/2$, and $N = 2^{12}$ blocks this already takes 3.5 days.

Π can be instantiated using a Feistel cipher with keys derived pseudorandomly from σ. If $8N$ is not an even power of 2, then we can use a standard cycle-walking technique. find the smallest k such that $8N \in [2^{2k-1}, 2^{2k}]$ and implement a Feistel cipher F over $2k$-bit block size. On each input $x \in [8N]$ iterate $F(x)$ until reaching the first output that is an integer in the range $[8N]$. The inverse permutation Π runs the inverse Feistel cipher, employing the same technique. If y is the first integer in $[8N]$ output by iterating F on x then it is easy to see that x will be the first integer in $[8N]$ output by iterating F^{-1} on y.

In what follows, DAG.Enc is the encoding subroutine DRG.Enc called by PoRep.Replicate in the DRG-PoRep construction, but replacing the function DRG.Parents with a new function DAG.Parents. This new function calls DRG.Parents but also adds "expander" edges. Furthermore, there is an "even" and "odd" mode of selecting the expander edges. The expander edges added in the odd mode are equivalent to reversing the edges of the even mode and renumbering the nodes in the reverse direction. That is, there is a directed edge (i, j) in the even mode graph if and only if there is a directed edge $(N - j + 1, N - i + 1)$ in the even mode graph. The DRG edges are sampled the same way in both graphs. DAG.Enc$(\vec{x}, m, N, \sigma, b)$ makes calls to DAG.Parents(N, σ, i, b) for $i \in [N]$. In pseudocode, the function DAG.Parents operates as follows:

```
DAG.Parents($N, \sigma, i, b$){
    $V := \{v_1, ..., v_d\} \leftarrow$ DRG.Parents$(N, \sigma, i)$
    $W := \emptyset$
    for $k = 1$ to $8$ :
     case $b = 0$ :
        $(j, k') \leftarrow \Pi(\sigma, (i, k))$;   $(j', k'') \leftarrow \Pi^{-1}(\sigma, (i, k))$
        if $j < i$ then $W := W \cup \{j\}$
        if $j' < i$ then $W := W \cup \{j'\}$
     case $b = 1$ :
        $(j, k') \leftarrow \Pi(\sigma, (N - i + 1, k))$;   $(j', k'') \leftarrow \Pi^{-1}(\sigma, (N - i + 1, k))$
        if $N - j + 1 < i$ then $W := W \cup \{N - j + 1\}$
        if $N - j' + 1 < i$ then $W := W \cup \{N - j' + 1\}$
return $W \cup V$}
```

PoRep.Setup$(\lambda, \kappa, T) \rightarrow pp$**:** The setup obtains as input security parameters λ, κ, as well as the delay parameter T and runs $pp_{\mathsf{vde}} \leftarrow$ VDE.Setup(1^λ). This determines a block size m and $\mathcal{M} = \{0,1\}^m$. The setup then runs $pp_{\mathsf{vc}} \leftarrow$ VC.Setup$(1^\lambda, N_{max}, \mathcal{M})$ where N_{max} is the maximum supported data length. Finally the setup also defines two integers $\ell_1 = \ell_1(\lambda)$ and $\ell_2 = \ell_2(\kappa)$. The setup outputs $pp = (pp_{\mathsf{vde}}, pp_{\mathsf{vc}}, m, \ell_1, \ell_2)$. We fix the number of layers L in the construction.

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 $H(id||\tau_D) = \sigma$.
2. Parse $\tilde{D}$ as data blocks $\vec{d} = (d_1, ..., d_N)$, each $d_i \in \{0,1\}^m$.

Initialize $\vec{x} = (x_1, ..., x_N)$ consisting of N data blocks where initially $x_i = d_i$. Define $\mathsf{swap}(\vec{x}) = (x_N, ..., x_1)$, which reverses the indexing of the elements. Iterate $\mathsf{DAG.Enc}$ L times over $\vec{x}$ as follows:

```
$\mathsf{StackedDRGEnc}(\vec{d}, m, N, \sigma)\{$
  **for** $i = 1$ *to* $L$ :
    $\vec{x}^* := \mathsf{swap}(\vec{x})$
    $\vec{x} := \mathsf{DAG.Enc}(\vec{x}^*, m, \sigma, i \text{ \% } 2)$
    $\Phi_i \leftarrow \mathsf{VC.Com}(pp_{\mathsf{vc}}, \vec{x})$
  $R \leftarrow \vec{x}$
  $\Phi \leftarrow \mathsf{VC.Com}(pp_{\mathsf{vc}}, \Phi_1, ..., \Phi_\ell)$
  **return** $R, \Phi\}$
```

When implementing $\mathsf{StackedDRGEnc}$, the values x_i can be updated in place, so only a single buffer of size N blocks is needed. Some extra storage is needed to store the vector commitment, however this can be made relatively small using time/space tradeoffs discussed below.

3. Use H to derive a challenge vector for each jth level as $\rho^{(j)} = (\rho_{1,j}, ..., \rho_{\ell_1,j})$ where $\rho_{i,j} = H(id||\Phi||i||j)$.

4. Rerun[10] $\mathsf{StackedDRGEnc}$ and on the jth iteration let $(c'_1, ..., c'_N)$ denote the output labels on the jth inputs $(c_1, ..., c_n)$.

   Compute vector commitment opening proofs on the indices specified by the challenges $\rho^{(j)}$:

   (a) Set $C_j^{\mathsf{nodes}} = (c_{\rho_{1,j}}, ..., c_{\rho_{\ell_1,j}})$.

   (b) For each i set $C_j^{\mathsf{parents}}(\rho_{i,j}) = (c'_{v_1}, ..., c'_{v_d})$ where $\{v_1, ..., v_d\} \leftarrow \mathsf{DAG.Parents}(m, \sigma, \rho_{i,j}, i \bmod 2)$. Let $\mathsf{parents}(\rho_{i,j}) = (v_1, ..., v_d)$.

   (c) Set $C_j^{\mathsf{pred}} = (c_{\rho_{1,j}}, ..., c_{\rho_{\ell_1,j}})$.

   Let $C^{\mathsf{nodes}} = (C_1^{\mathsf{nodes}}, ..., C_L^{\mathsf{nodes}})$, and $C^{\mathsf{parents}} = \{C_j^{\mathsf{parents}}(\rho_{i,j})\}_{j\in[L],i\in[\ell_1]}$, and $C^{\mathsf{pred}} = (C_1^{\mathsf{pred}}, ..., C_L^{\mathsf{pred}})$.

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

```
**for** $j = 1$ *to* $L$ :
  $\Lambda_j^{\mathsf{nodes}} \leftarrow \mathsf{VC.Open}(pp_{\mathsf{vc}}, C_j^{\mathsf{nodes}}, \Phi_j, \rho)$
  $\Lambda_j^{\mathsf{pred}} \leftarrow \mathsf{VC.Open}(pp_{\mathsf{vc}}, C_j^{\mathsf{pred}}, \Phi_j, \rho)$
  **for** $i = 1$ *to* $\ell_1$ :
    $\Lambda_{i,j}^{\mathsf{parents}} \leftarrow \mathsf{VC.Open}(pp_{\mathsf{vc}}, C^{\mathsf{parents}}(\rho_{i,j}), \mathsf{parents}(\rho_{i,j}))$
```

6. Output R, and
   $\mathsf{aux} = \Phi_1, .., \Phi_L, C^{\mathsf{nodes}}, C^{\mathsf{parents}}, C^{\mathsf{pred}}, \{\Lambda_j^{\mathsf{nodes}}\}_{j\in[L]}, \{\Lambda_j^{\mathsf{pred}}\}_{j\in[L]}, \{\Lambda_{i,j}^{\mathsf{parents}}\}_{j\in[L],i\in[\ell_1]}$.

[10]The prover can of course choose between using a larger buffer (up to NL) and re-running the $\mathsf{StackedDRGEnc}$ computation once.

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

$\textsf{PoRep.Prove}(R, aux, id, r) \rightarrow \pi$: The input is id, the $\textsf{aux}$ output of replicate, challenge vector $\vec{r} = (r_1, ..., r_{\ell_2})$, and the replica $R = (c_1, ..., c_N)$. Set $\vec{c} = (c_{r_1}, ..., c_{r_{\ell_2}})$. Compute $\Lambda \leftarrow \textsf{VC.Open}(pp_{\textsf{vc}}, com_R, \vec{c}, \vec{r})$. Output $\pi = \Lambda_{\textsf{ret}}, \vec{c}$.

$\textsf{PoRep.Verify}(id, \tau_D, r, aux, \pi)$: Parse the input $\textsf{aux}$ as $\Phi_1, ..., \Phi_L$, $C^{\textsf{nodes}}$, $C^{\textsf{parents}}$, $C^{\textsf{pred}}$, $\{\Lambda_j^{\textsf{nodes}}\}$, $\{\Lambda_j^{\textsf{pred}}\}$, $\{\Lambda_{i,j}^{\textsf{parents}}\}$. Parse $\pi = \Lambda_{\textsf{ret}}, \vec{c}$.

1. (This is only done the first time, otherwise skip to 2). Derive $H(id||\tau_D) = \sigma$, and $\rho_{i,j} = H(id||\Phi||i||j)$ for each $i \in [\ell_1]$ and $j \in [L]$. Derive for each i, j the set $\textsf{parents}(\rho_{i,j}) \leftarrow \textsf{DAG.Parents}(m, \sigma, \rho_{i,j})$. Run verifications of all the vector commitment openings on $C^{\textsf{nodes}}$, $C^{\textsf{parents}}$, and $C^{\textsf{pred}}$.
2. Second verify the proof π. Compute $b \leftarrow \textsf{VC.Verify}(pp_{\textsf{vc}}, \vec{c}, \vec{r}, \Lambda_{\textsf{ret}})$. Output b.

Vector commitment overhead 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 [25] these 2^k reads are sequential memory reads, which in practice are inexpensive compared to the random reads for challenge labels. Similar time/space tradeoffs are possible with RSA vector commitments.

Reducing number of challenges In the description of the protocol above we set the number of non-interactive challenges ℓ_1 to be the same at every level, for a total of $\ell_1 L$ challenges. Since $\ell_1 = O(\lambda/\epsilon)$ to achieve ϵ-rational-security this results in proof size of $O(\lambda L/\epsilon)$. However, we can actually decrease the number of challenges between levels by a multiplicative factor, and provided that the number of challenges remains above a certain threshold. In particular, letting $\ell_1^{(i)}$ denote the number of challenges for the ith level, we prove security with $\ell_1 = \ell_1^{(L)} = O(\lambda/\epsilon)$ and $\ell_1^{(i)} = min(20, (2/3)\ell_1^{(i+1)})$, although the analysis could be tightened for better parameters. The total number of challenges is therefore $O(\lambda/\epsilon)$ because $\sum_{i=1}^{L} \ell_1^{(i)} \leq 3\ell_1$.

Data extraction The data extraction is less efficient compared to DRG-PoRep (which coincide with $L = 1$). There is a still a large asymmetry between the replication time and data extraction due to the asymmetry of VDE.Enc and VDE.Dec, however this difference is reduced as the file size grows larger. The extraction can still be highly parallelized allowing for a large speedup given sufficiently many parallel cores.

Proof size estimates The online proof size is comparable to the proof size of the DRG-PoRep, although now that an arbitrarily small ϵ-rational replication is actually achievable (also ϵ space gap) it will be necessary to set ℓ_2 appropriately in the online proof. As in Basic-DRG-PoRep, to verify that the prover can still retrieve an ϵ fraction of the commitment with soundness μ requires $\ell_2 > \log(\mu)/\log(1-\epsilon)$. For example, if $\epsilon = 0.01$ and $\mu = 1/3$ then $\ell_2 = 109$. With 16-byte block sizes this is 1.6KB. (Considering the file may be up to terabytes in size this is highly reasonable). A SNARG proof can also be used to compress the online proof size, and in this case would have multiplicative complexity around 10 million gates using a Merkle commitments with the Jubjub pedersen hash, and can be generated in 1.5 minutes on 16 parallel cores.

The non-interactive proof included in aux is now somewhat larger than in DRG-PoRep because of the increase in levels, i.e. it is a factor $L = O(\log(1/\epsilon))$ larger without considering the optimization of reducing the number of challenges between levels as described above. When applying this optimization, then asymptotically it is only $O(1/\epsilon)$ as $\epsilon \to 0$. This can still be compressed with SNARGs, but now has a larger circuit complexity as well and will thus take longer to compute. Without using SNARGs and instantiating the vector commitments with RSA vector commitments the bottleneck in the proof size is the labels of the challenges. Let $\ell_1^* = max(20L, 3\ell_1)$. The proof size is approximately $\ell_1^* \cdot d \cdot m$ bits where m is the block size and $\ell_1 = O(\lambda/\epsilon)$. Concretely, for soundness 2^{-8} and $\epsilon = 0.03$ then according to our analysis we can set $\ell_1 = 3/0.01 = 300$ and $L = 10$ for total $\ell_1^* = 900$. (This soundness level makes the assumption that any rational attacker will not repeat its initialization more than 2^8 times in order to save $\epsilon = 0.02$ space). With $m = 128$ and $d = 13 = 5 + 8$ this results in a proof size of approximately 187 KB, independent of the data input size. This is still reasonable consider the data input sizes could range up to gigabytes or terabytes.

One observation is that the edges of the DRG can tolerate more error than the expander edges between layers given a suitably strong DRG construction. For example, if the DRG is depth robust in 70% subgraphs and at most 10% of the nodes in the graph contain errors, then a subgraph on 80% of the nodes will still be depth robust. On the other hand, if only an ϵ fraction of the dependencies between levels (i.e. the predecessors) are enforced then ϵ-rationality takes a direct hit. Likewise, the expander edges are what guarantee expansion of dependencies between levels. Therefore, we could reduce the proof size by adjusting the number of queries checking the DRG edges vs the expander edges. Reducing this can makes a significant difference as each time we check the DRG edges we need to open d labels.

4 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 implication of a larger ϵ is that the erasure code will have to tolerate up to an ϵ fraction deletion of the data, which will require an erasure code blowup factor of $r = 1/(1-\epsilon)$ (note that $r < 2$ for $\epsilon < 1/2$). The degree impacts the PoRep proof size and verification complexity, which is $O(\lambda d)$. On the other hand, Stacked-DRG-PoRep only requires an $(n, 0.80, \beta, d)$ DRG for some constant β and degree d.

Explicit Depth Robust Graphs Erdős et. al. [24] 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 [23], 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. [22] 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. [5] 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. [4] 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).

4.1 Bucket Sampling

The starting point of our DRG sampling algorithm is the algorithm of Alwen et. al. [4], 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 $u \overset{R}{\leftarrow} B_i$ is selected. The edge (u, v) is added to the graph.

```
BucketSample$[n]$\{
$V := [n];\ E := \emptyset$
for $v = 2, ..., n$ :
    $E := E \cup \{(v-1, v)\}$
    $i \overset{R}{\leftarrow} \{1, ..., \log_2 v\};\ u \overset{R}{\leftarrow} B_i(v)$
    $E := E \cup \{(u, v)\}$
return$(V, E)$\}
```

```
BucketSample$[n, m]$\{
$(V, E) \leftarrow$ BucketSample$[nm]$
$W := [n];\ E' := \emptyset$
for$(i, j) \in E$ :
    $u_i \leftarrow i \% n;\ u_j \leftarrow j \% n$
    $E' := E' \cup \{(u_i, u_j)\}$
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 2 (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 $D_m(S) = \{x \in G : \exists u \in S s.t. 0 \leq u - x \leq m\}$ 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. [4] 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., $(1-\beta)/(c_1c_2)$) 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 1. *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. □

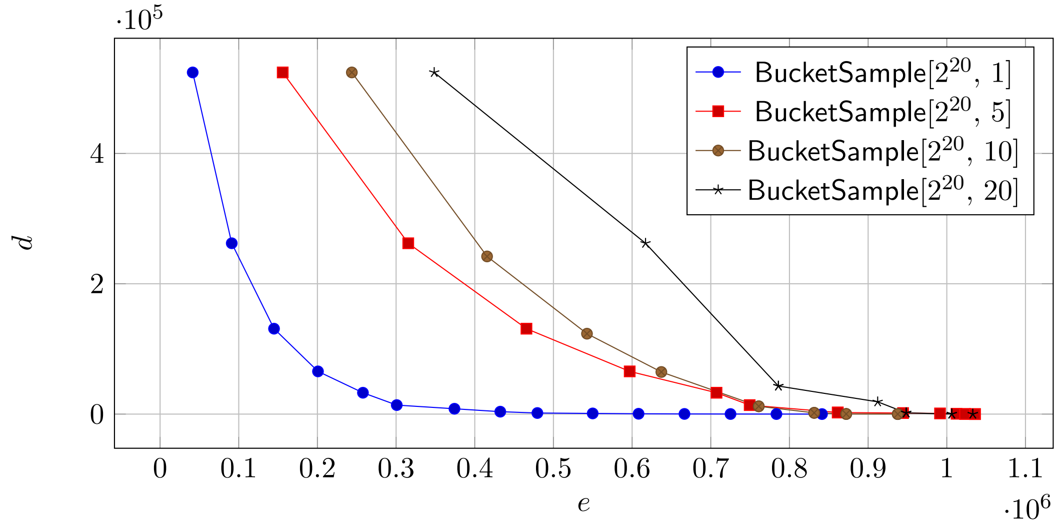

Figure 4.1: Results of the attacks against BucketSample$[n, m]$ for m = 1, 5, 10 and $n = 2^{20}$. We plot on the y-axis for each value of $e < n$ on the x-axis the smallest depth among any of the subgraphs of size $n - e$ that any of the attacks were able to find. For example, with BucketSample$[n, 5]$ 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 [4], 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 [29] for locating a depth reducing set, shown below in Figure 4.1. 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.

References

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

[2] 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.

[3] 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.

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

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

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

[7] Leonid Alexandrovich Bassalygo. Asymptotically optimal switching circuits. In *Problemy Peredachi Informatsii*, 1981.

[8] 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.

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

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

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

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

[13] Ben Fisch. Poreps: Proofs of space on useful data. Cryptology ePrint Archive, Report 2018/678, 2018. `https://eprint.iacr.org/2018/678`.

[14] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password scrambler. *IACR Cryptology ePrint Archive*, 2013.

[15] Dan Boneh Henry Corrigan-Gibbs and Stuart Schechter. Balloon hashing: a provably memory-hard function with a data-independent access pattern. In *Asiacrypt*, 2016.

[16] 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.

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

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

[19] Sergio Demian Lerner. Proof of unique blockchain storage, 2014. `https://bitslog.`

`wordpress.com/2014/11/03/proof-of-local-blockchain-storage/`.

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

[21] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*. ACM, 2013.

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

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

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

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

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

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

[28] Uwe Schöning. Better expanders and superconcentrators by kolmogorov complexity. In *SIROCCO*, 1997.

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

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