PIEs: Public Incompressible Encodings for Decentralized Storage

Ethan Cecchetti
Cornell University; IC3†
ethan@cs.cornell.edu

Ben Fisch
Stanford University
benafisch@gmail.com

Ian Miers
Cornell Tech; IC3†
imiers@cornell.edu

Ari Juels
Cornell Tech; IC3†
juels@cornell.edu

†Initiative for CryptoCurrencies & Contracts

Abstract

We present a provably secure approach to proving file replication (or other erasure coding) in distributed storage networks (DSNs). Storing multiple copies of a file F is essential in DSNs to ensure against file loss in the event of faulty servers or corrupt data. The *public* nature of DSNs, however, makes this goal challenging. Files must be encoded and decoded using public coins—i.e., without encryption or other secret-key operations—and retention of files by servers in the network must be publicly verifiable.

We introduce and formalize the notion of a *public incompressible encoding* (PIE), a primitive that supports file-replication proofs in the public setting. A PIE enables public verification that a server is storing a replicated encoding G of a target file F, and has not compressed G to save storage. In a DSN with monetary rewards or penalties, PIEs can help ensure that economically rational servers store G and thus replicate F honestly.

Our PIE constructions are the first to achieve experimentally validated *near-optimal performance*—only a factor of 4 from optimal by one metric. They also support *fast decoding* which no other comparable construction does—as much as a factor of over 750 in one experiment. PIEs additionally meet the critical security requirements for DSNs and related applications: they preclude demonstrated attacks involving parallelism via ASICs and other custom hardware. They achieve all of these properties using a graph construction we call a *Dagwood Sandwich Graph* (DSaG) that involves a novel interleaving of depth-robust graphs and superconcentrators.

PIEs' performance makes them appealing for DSNs, such as the proposed Filecoin system, and other challenging file-storage needs in public settings. Conversely, their near optimality provides realistic measures of the (considerable) practical costs and limitations of DSNs and other applications.

1 Introduction

The world's data storage demands are ballooning, with annual growth rates of 42% and a projected 50 zettabytes required by 2020 [50]. Supply, however, is lagging [22], while much of the world's hard drive space sits idle. This has led to the rise of *decentralized storage networks* (DSNs) such as Sia [58], Storj [59], MaidSafe, and Filecoin [47] that store data on unused devices in peer-to-peer systems. DSNs meet not only a universal demand for storage, but also specific needs in blockchain systems. Ethereum's heavily replicated blockchain, for example, recently grew from 20 GB to over 100 GB in a year [27], prompting calls for new decentralized storage architectures [15].

All DSNs pose a fundamental technical challenge: *proving data is stored robustly*. DSNs need to assure users that their files are not just stored, but stored *redundantly*—with replication or other erasure coding—to prevent data loss resulting from hardware and software failures or lost peers. In conventional cloud storage systems, users simply trust providers to faithfully replicate files (e.g., Amazon S3 stores three replicas). Decentralized systems, in contrast, involve *untrusted* peers that must *prove* they have done so.

Well-established *proof of storage* techniques such as Proofs of Retrievability (PoRs) [38, 53] and Proofs of Data Possession (PDPs) [7] allow an untrusted provider to prove retention of a file F efficiently to a client. Unfortunately, these techniques do not enable a client or verifier to distinguish between an honest provider robustnessly storing three copies of F and a cheating provider that stores a single, brittle copy. Proving file replication thus requires a different set of techniques. This is particularly true given that proposed DSNs are designed to reward replication, creating a monetary incentive for providers to cheat and save storage by falsely claiming to replicate files.

1.1 The Challenge of Proving Replication

It is straightforward to prove replicated storage of a file F in a restricted *trusted-encoder*, *private-reader* setting where the file owner/client can use secret keys to encode and retrieve F. To store three copies, the client simply generates secret keys $(\kappa_1, \kappa_2, \kappa_3)$, encrypts F under each key respectively, and uploads the resulting ciphertext triple $G = (C_1, C_2, C_3)$. The provider can prove to the client using a PoR/PDP on G that it is storing all copies. Use of encryption prevents the provider from cheating and discarding a file copy or compressing file contents—but it also means only the owner can retrieve F. Systems such as Sia [58] and Storj [59] support this approach.

Proving replication is also possible in a *trusted-encoder*, *public-reader* setting. Existing mechanisms [21, 57] show how an encoder (e.g., file owner) can perform a private-key operation using a trapdoor one-way function (RSA) to encode individual replicas of F. Any entity can recover F from this encoding with a private key. Additionally, anyone can verify storage of any replica using a publicly verifiable PoR/PDP.

The biggest challenge arises in the *untrusted-encoder*, *public-reader* setting—which we call the *public* setting for simplicity. In this setting, any (untrusted) entity can apply a publicly specified *encoding* function ENCODE to replicas of any file F, yielding a redundant encoding G. (For example, $G = \text{ENCODE}_1(F) \parallel \text{ENCODE}_2(F) \parallel \text{ENCODE}_3(F)$ for a set of encoders $\{\text{ENCODE}_i\}$.) The correctness of these encodings should be *publicly* verifiable. Anyone should be able to verify a proof of storage (e.g., PoR) for G, and anyone should be able to decode a sufficiently intact G to recover F (via a function DECODE). Critically, in this setting, *no operation by any entity requires secret keys: all coins are public.*

The public setting is essential for several applications. In Filecoin, for instance, miners prove replicated storage of public files [47]. They have an incentive to cheat in order to reduce their storage costs, and thus are untrusted encoders. Similarly, smart contracts cannot manage secret keys, so they must store data via untrusted encoders.

Proving replicated file storage in the public setting is difficult, though. To begin with, any proof of storage of G must seemingly rely on *timing assumptions*.[1] Since ENCODE is public, a cheating prover given arbitrary time to respond to challenges can just recompute $G = \text{ENCODE}(F)$. Constructing a function ENCODE that imposes a strong lower bound on a cheating prover's response time is thus a major technical challenge. Additionally, in the public setting it is actually impossible to *guarantee* that a prover is really storing each encoded copy G_i of the file F in a truly replicated way [29]. For example, the prover might store an encryption of G with key unkonwn to the client. The ENCODE function can at best ensure that there is no economic incentive to cheat, i.e. reduce storage cost.

As we explain in Section 2.2, previous attempts to construct a practical encoder have fallen short of achieving these requirements in a provably secure way without relying on implausible assumptions.

[1] Timing assumptions may not be needed. For example, if encoding uses very long private keys, an economically rational server might be unwilling to store these keys and unable to recompute encodings on the fly. The feasibility of such approaches is an open research question.

1.2 A Public Incompressible Encoding (PIE)

We introduce a provably secure technical tool to enable proofs of replication in the public setting: a *public incompressible encoding* (PIE). Our construction has two main features that distinguish it over prior work. First, we show that it requires computation *within a small constant factor of optimal*. Second, it supports *fast decoding*, making it particularly appealing for DSNs, which are write-once, read-many systems. This fast decoding can take the form of parallelization in a basic construction or leverage asymmetric-speed permutations, such as Sloth [40], for strictly faster serial decoding. Together, these two properties make our construction among the most appealing for practical applications.

A PIE is a type of *proof of space* [6, 25] with the special requirement that it can encode files [29, 45]. Informally, it prevents an adversary from undetectably compressing G by more than a tiny amount. Specifically, suppose an adversary stores a compressed representation G' such that $|G'| \leq (1-\epsilon)|G|$. Our main result states that any adversary challenged to produce a randomly selected block of G must perform a long sequential computation with probability at least ϵ (minus a negligible term). The resulting delayed response will be detectable by the verifier. We can boost soundness in the standard way with multiple queries.

In DSNs, a provider is monetarily rewarded for periodically proving retention of an encoded file G and delivering it on demand. As we argue, in such settings a PIE is sufficient to ensure that an *economically rational* provider will correctly and fully store G—our main objective.

Our Construction. Our proposed PIE ENCODE is a graph-based file transformation. Specifically, it depends on a new construct we call a *Dagwood Sandwich Graph* (DSaG).[2] A DSaG is an iterated interleaving of two graph components. The first is a *depth-robust graph* (DRG)—intuitively, a directed acyclic graph that retains a long paths even if an adversary removes a large number of nodes. DRGs are commonly used in memory-hard hash function designs [2–4, 12] and have been used to enforce long sequential computations [42], including in proofs of space [25, 30, 45], a purpose we exploit in our DSaG construction. The second component is a *superconcentrator* [56]: a graph with many vertex-disjoint paths among input and output nodes, creating a strong dependency of any output on all inputs. We show a good practical choice to be a doubling of the classic butterfly graph [20]. By sandwiching these two components and iterating, we can prove that any attempt at compression, such as discarding a block of G and attempting to recompute it on the fly, forces a storage provider to recompute blocks of G in an expensively slow and detectable fashion.

We show experimentally that, for a variety of realistic and highly tunable concrete timing bounds, our construction's encoding performance is only a factor of 4 from the theoretical optimum by a metric we call the *security efficiency ratio* (SER). Fast serial decoding via Sloth [40] yields an order of magnitude speedup over encoding for typical parameters. Our construction operates on files as small as tens of kilobytes and achieves high parallelism and thus efficient scaling on larger files.

Limitations. The near-optimality of our construction (under the SER) *also* supports negative results. It establishes performance bounds on general-purpose, public-setting DSNs such as Filecoin and reveals notable and fundamental practical limitations. These include very high computation costs that may render such systems uneconomical for most data.

There are limited-purpose public-setting DSN applications where PIEs may have strong practical promise. These include proposed schemes for storage of Ethereum blockchain data [1], minimizing storage in permissioned blockchains via verifiable erasure coding across nodes and, in cloud systems, enforcing file properties such as at-rest encryption, watermarking of files, binding of licenses to data, etc. [57].

[2] A Dagwood is a many-layered sandwich made famous in the classic cartoon strip Blondie. It is visually evocative of our construction.

Paper Organization and Contributions

After discussing background on DSNs and prior approaches in Section 2, we present our main contributions:

- *Public Incompressible Encodings (PIEs)*: We define the security of a PIE in Section 3. Our PIE construction follows in Section 4, where we introduce the *Dagwood Sandwich Graph* (DSaG), our novel graph construction.
- *PIE applicability:* In Section 5, we discuss how PIEs' near-optimal performance and fast decoding and offer insight into lower-bound costs of schemes such as the Filecoin DSN, as well as resulting considerations regarding DSNs practicality and application suitability.
- *Implementation and experiments:* We eliminate unnecessary overhead in Section 6 and define the security efficiency ratio (SER), our main efficiency metric, in Section 7. In Section 8 we report experimental performance results.

Section 9 discusses related work and Section 10 concludes.

2 Background

We now give an overview of PIE use in DSNs and challenges in their construction shown by previous work.

2.1 Using Incompressible Encodings in DSNs

Incompressible encodings do not, by themselves, solve the problem of ensuring file storage robustness in DSNs. They do not spread storage across multiple nodes or ensure data availability. They do, however, provide a necessary component for building such a system by enabling *detection* of malicious nodes that use less storage than claimed.

Recall that DSNs aim to ensure that a file F is stored redundantly, i.e., with erasure coding or replication. The idea is that F should still be recoverable even if some copies or pieces are lost or corrupted. In a public DSN, storage servers are considered potentially untrustworthy. To encourage good behavior, they are periodically compensated for storing data. As a result, they have an incentive to reduce storage costs wherever possible: the more data a server can claim to store, the more revenue it can generate. The best way to reduce storage costs, of course, is for a server to simply not store the data for which it is responsible, but this is only beneficial if it can escape detection of this bad behavior. Proofs of storage [7, 38, 53], as noted above, can detect when a server fails to store a target file F. But they cannot detect failures to store a file F redundantly.

Consider, for instance, a setting in which three servers S_1, S_2, and S_3 are each supposed to store a copy of F. What is tricky is that these servers can *collude* undetectably. S_1 can correctly store one copy of F, while S_2 and S_3 discard theirs. When challenged to provide a copy of F, S_2 or S_3 can simply access the copy held by S_1. Provided this single copy remains available, all servers can furnish valid proofs of storage. The problem, of course, is: if the single copy held by S_1 is lost or damaged, F will be unrecoverable.

An alternative method of detecting deduplication is to *time* the responses of servers challenged to prove possession of F. If S_2 must obtain F from S_1, it will respond to challenges more slowly than S_1, which can access F directly. Such timing, though, can be unreliable given variance in network latency. Public incompressible encodings (PIEs), our focus here, address this problem by using a slow file transformation that makes server response times tunable; larger times subsume variable network latency. PIEs are designed so that a malicious server that has deduplicated data will take much longer to respond than an honest server.

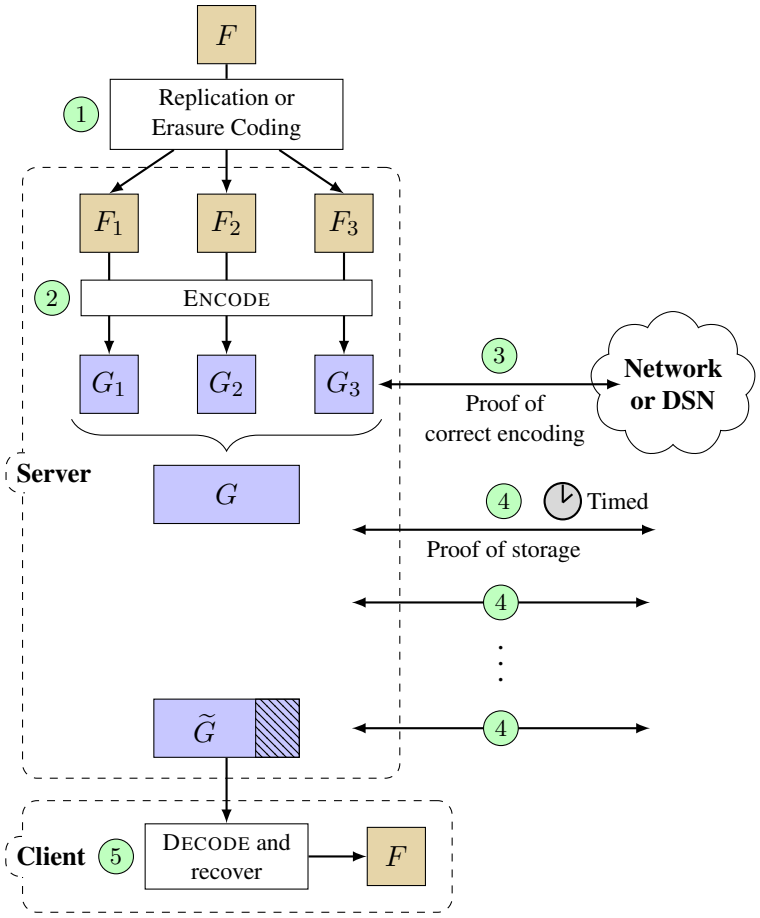


Figure 1: The basic structure of a DSN using a PIE to ensure data robustness. (1) First, file F is replicated or erasure-coded to ensure robustness. (2) The server runs this robust data through ENCODE yielding incompressible representation G. (3) The network/DSN queries the server to prove the data was encoded properly. (4) The network/DSN repeatedly queries the server and times responses to ensure the data is properly stored. (5) Upon request, a client can download and decode the stored data to recover F. If some of the stored data G is lost or corrupted, resulting in only $\widetilde{G}$ being available, the DSN can detect this and the client can still recover F.

The general procedure for applying a PIE is as follows. The target file F is first transformed into a redundant state, for example replicated as $(F_1 \,\|\, F_2 \,\|\, F_3)$, where $F_i = F$. This redundant version of F is then encoded using the PIE into a representation G, e.g., $G = (G_1 \,\|\, G_2 \,\|\, G_3)$. Of course, the DSN needs to ensure that G is correctly encoded. Once G is stored, DSN servers are challenged periodically to prove possession of G. If G is fully stored, then F is still retrievable even given damage to G. Consequently, with this arrangement a DSN can verify the erasure-resistance of its data. We illustrate these steps in Figure 1. Other applications of PIEs, such as proving files bear watermarks [57], look similar.

Financial penalties can be levied against servers caught *not* storing G (see, e.g., [47].) If they are set appropriately, deleting even a small fraction of G would incur a substantial financial risk relative to the saved storage cost, while storing only a small fraction of G would mean an overwhelming probability of detection. A rational server would thus be incentivized to store G correctly and in its entirety.

The problems of how to set financial penalties in a DSN, as well as the orthogonal problem of ensuring that G is *distributed across servers or storage devices* (see, e.g., [10, 14]) are out of scope for this work.

Ensuring Correct PIE Computation. In the public setting we consider, no entity can be trusted to compute G correctly. A PIE itself does not ensure correct encoding—only incompressibility. As noted, a DSN would need to additionally ensure that G is correctly computed from F.

In one sense this problem is trivial. If anyone can encode or decode the data, then every node in a DSN can simply check than $G = \text{ENCODE}(F)$. Unfortunately, due to the inherent slowness of PIE constructions, having every node in the DSN validate the correct encoding in this way would incur prohibitive computational costs for the network.

There are multiple ways to address this concern depending on the setting. In a permissioned or hybrid consensus system [43], which selects small committees presumed to contain a quorum of honest nodes, a subset of nodes can verify correct file encoding. In a permissionless settings, an alternative approach is not to verify correct encodings, but to create financial incentives to deter misbehavior. We briefly sketch a "positive" scheme where the encoder produces proof that an encoding is *correct*; this scheme detects misbehavior with high but not overwhelming probability. We also describe a "negative" scheme in which any entity can generate a compact proof that shows with overwhelming probability that an encoding is *incorrect*. These approaches are complementary and further improve security when combined.

The "positive" proof approach relies on the fact that our ENCODE function is defined by a graph. During encoding, it is possible to non-interactively sample vertices in the graph and prove that they were correctly computed with respect to their parents. Using this approach, an encoder can prove that a large percentage of the vertices were correctly computed—the usual approach in proofs of space [25, 48]. A small number of incorrect vertices could escape detection, but given a suitable penalty, requiring a positive proof could deter a rational encoder from cheating on more than a small fraction of vertices. In theory, the encoding can also be proven correct via non-interactive verifiable computation [34], such as SNARKs.

The "negative" proof approach relies on our observation that anyone can recompute ENCODE or DECODE and thus check that $G = \text{ENCODE}(F)$. It is possible then to construct a compact proof that shows with overwhelming probability that a particular encoding is *incorrect*. This proof can be constructed using a SNARK, or interactively using more straightforward binary search techniques and refereed delegation of computation [8, 16], such as those popularized by Truebit for detecting incorrect smart contract computation [55]. With this approach, the network can financially reward whistleblowing, incentivize users to verify encodings and submit proofs when they are incorrect, and impose financial penalties for proven-incorrect encodings.

As the right mechanism for verifying correct encoding is context-dependent and technically orthogonal to PIE construction and use, we omit further details.

2.2 Previous Approaches to PIEs

To understand the challenges of constructing a PIE, we briefly describe previous works with similar guarantees.

Proofs of Space. A PIE is closely related to a *Proof of Space* (PoS) [25], an interactive protocol in which a prover publicly commits to an amount of space and then responds to challenges. Soundness of a PoS requires that the prover fail with high probability if it is using significantly less space than claimed. In fact, the formal soundness definition of a PoS is very similar to the incompressibility requirement of a PIE.

PoS soundness and incompressibility differ in two key ways. First, most PoS definitions account for the possibility that a prover may cheat while initializing its storage, while we separate the problem of proving the incompressibility of a correct encoding from verifying the encoding's correctness. Second, a PoS adversary might be able to save a constant fraction of the space promised and evade detection with overwhelming probability. PIEs do not allow this. This makes a PIE more similar to a *tight* PoS [30, 45], in which an adversary saving any ϵ fraction of space must fail a random challenge with probability proportional to ϵ. Until

recently, no PoS construction provided proof that an adversary must use even half of the promised storage.

The similarity to a tight PoS is so strong, in fact, that any tight PoS can be used to construct a PIE. A generic construction of Pietrzak [45], called a *proof of catalytic space*, simply derives a PoS S of the same size as F and outputs $F \oplus S$. Inverting this encoding, however, requires re-deriving S making it extremely inefficient to recover F. To improve on this, efficient data extraction is a major goal of this work.

Replicated Storage. Several PIE constructions (under various names) aim to prove replicated storage. Lerner [41] suggests using "time-asymmetric encodings," such as a Pohlig-Hellman cipher [46] based on modular exponentiation, to apply a slow transformation to each input file block using a unique identifier as a key. Boneh et al. [11] generalize this construction using *decodable verifiable delay functions*. Intuitively, this yields a sound PIE, as re-deriving any block of the encoding is slow. Unfortunately, the encoding time for these constructions scales poorly with file size, resulting in a poor *security efficiency ratio* (SER)—our efficiency metric defined in Section 7.

The Filecoin [47] project introduced the term *proof of replication* (PoRep) to codify the desired properties of a primitive for their DSN. They propose using PoReps to both encode data and provide Sybil resistance for a blockchain by replacing proof-of-work with proof-of-space. Their initial technical report [9] proposed a construction based on multiple chained (CBC) encryption passes over the entire file F under a published key κ, with block permutations interleaved between passes. Unfortunately, previous work [57] anticipates an attack against this scheme unless it uses a large number of passes. A cheating prover can checkpoint specific intermediate blocks of the encryption as "shortcuts" which it can use to quickly recompute discarded output. A checkpointing adversary can recompute outputs of a $\sqrt{d}$-pass CBC in at most d steps, so an encoding time of at least $n\sqrt{d}$ is required for a file of size n to achieve sequential security of d operations.

Hourglass functions [57] provably resist such "shortcuts" by applying published-key pseudorandom permutation (PRP) to pairs of blocks of F in a sequence determined by a butterfly network. Unfortunately, hourglass functions and related approaches [14] assume slow retrieval of F due to use of rotational hard drives. The proliferation of fast solid-state drives (SSDs) and monetary incentives to cheat in systems like Filecoin clearly invalidate this assumption.

Consequently, previous work offers no general, practical approach to remotely prove storage redundancy for a file while supporting efficient public decoding.

3 Security Definitions

Here we define a secure *public incompressible encoding* (PIE) scheme. An *encoding* is a function pair $(\textsc{Encode}, \textsc{Decode})$ such that for all files F and coins ρ,

$$\Pr[(G, \kappa) \leftarrow \textsc{Encode}(F\,;\,\rho) : \textsc{Decode}(G, \kappa) = F] = 1.$$

Here, κ is a key output by $\textsc{Encode}$ and stored with F for use by $\textsc{Decode}$. An encoding is *public* if the coins ρ are public.

Recall that our goal is to ensure that any adversary that uses a PIE to encode data must actually store as much data as claimed, regardless of data contents. We must guard against both on-the-fly re-encoding, and more subtle forms of cheating like storing intermediate values used to encode F.

We define what it means for a PIE encoding to be *incompressible* in terms of a game between the encoder and a public challenger. An adversary $\mathcal{A}$ chooses a set of (not necessarily distinct) files $\{F_i\}$. When the $\{F_i\}$ are encoded to $G = \{G_i\}$ under random coins, $\mathcal{A}$ is challenged to produce randomly selected blocks of G. The encoding is incompressible if $\mathcal{A}$ can only respond successfully to challenges (except with negligible

probability) by using at least $|G|$ storage, i.e. $\mathcal{A}$ cannot compress or deduplicate G. Such incompressibility is impossible to achieve in the public setting without further assumptions. For example, if $F = F_1 = F_2$, a cheating $\mathcal{A}$ could store only F, and respond to challenges by computing blocks of $G = (G_1, G_2)$ on the fly.

In practice we can get around this impossibility by ensuring that ENCODE is time consuming. Specifically, any cheating adversary must incur a detectably long delay before responding to challenges, even if it is allowed to use unbounded (feasible) parallel computation. Addressing parallel computation is important given current developments in so-called ASIC mining [54], which uses custom hardware to achieve modest improvements in sequential performance and drastically increased parallelism. This security notion has been used before in time-lock puzzles [49], proofs of sequential work (PoSW) [17, 42], verifiable delay functions (VDFs) [11], and more.

Modeling time-consuming computation is tricky as the computation may involve a mixture of heterogenous operations. Time-lock puzzles and related primitives are easiest to analyze when they involve an iterated atomic operation, such as a hash function, and their hardness can be measured as the minimum number of sequential invocations of this operation, modeled as queries to an oracle. Constructions of time-lock puzzles, PoSW, and PoS based on iterated calls to a hash function have been proven to require sequential work in the parallel random oracle model [42, 45].

We follow this approach and limit the number of times an attacker can execute an intentionally slowed key derivation function modeled as an oracle $\mathcal{O}$. We constrain the number of sequential calls to $\mathcal{O}$, defining a parallel oracle $\mathcal{O}_d^\star$ that will respond to any number of simultaneous queries to $\mathcal{O}$, but only $d-1$ sequential ones. In practice, adversaries who attempt to make more than $d-1$ sequential oracle queries would be caught by timing measures. Formally, we define $\mathcal{O}_d^\star$ as follows, initializing $c = 1$:

$\underline{\mathcal{O}_d^\star(x_1, \ldots, x_s)}$

if $c \geq d$ **then abort**
$c \leftarrow c + 1$
return $(\mathcal{O}(x_1), \ldots, \mathcal{O}(x_s))$

We now define security in terms of an adversary who must return encoded file blocks given limited storage and access to $\mathcal{O}_d^\star$. We let $|G|$ denote the length of a file G in units of λ-bit blocks and $G[i]$ be the ith such block. Definition 1 considers an adversary $\mathcal{A}$ who claims to store m honestly encoded files, but attempts to compress the combined encodings by a factor of ϵ. A PIE scheme is secure if $\mathcal{A}$ can do no better than simply discarding an ϵ fraction of the final blocks, hoping that the random challenge will not be in this set.

Definition 1 (Public incompressible encoding)**.** A public encoding algorithm ENCODE is *d-oracle incompressible* if for any compression factor $\epsilon < 1$ and any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $negl$ such that for all sets of files $\{F_i\}_{i=1}^m$,

$$\Pr\left[\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}(\lambda, \{F_i\}_{i=1}^m, \epsilon, d) = 1\right] \leq (1-\epsilon) + negl(\lambda),$$

where

$\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}(\lambda, \{F_i\}_{i=1}^m, \epsilon, d)$

$\{\rho_i\}_{i=1}^m \leftarrow\!\!\$\ \{0,1\}^{\lambda \times m}$

$G' \leftarrow\!\!\$\ \mathcal{A}_1^{\mathcal{O}}(\{(F_i, \rho_i)\}_{i=1}^m)$

for $i \in [1, m] : (G_i, \kappa_i) \leftarrow \text{ENCODE}^{\mathcal{O}}(F_i\,;\,\rho_i)$

$G := G_1 \,\|\, \cdots \,\|\, G_m$

$j \leftarrow\!\!\$\ [1, |G|]$

$blk \leftarrow\!\!\$\ \mathcal{A}_2^{\mathcal{O}^\star_d}(j, G', \{(\rho_i, \kappa_i)\}_{i=1}^m)$

return $\left(\frac{|G'|}{|G|} \leq 1 - \epsilon\right) \wedge (blk = G[j])\,.$

Challenges to $\mathcal{A}_2$ model the online portion of a real-world protocol in which a challenger can institute a configurable timing bound on an adversarial storage server. To realize this, it suffices to assume only that inherently sequential random oracle queries are slow. This allows us to configure a bound and avoids relying on questionable timing assumptions about rotational hard drive latency [57] or other explicit timing. $\mathcal{A}_1$ is only bounded by standard polynomial time, as it represents an attacker's offline attempt to compress outputs.

Multiple Queries. This definition allows only a single query to $\mathcal{A}$, but any practical usage would increase soundness through multiple queries. To avoid concerns of amortization across challenges, we consider a definition that allows multiple challenges and prove it equivalent in Appendix A.

4 Building a PIE

Now that we have formally defined the security of PIEs, we need to construct one. To satisfy our security definition and catch a cheating attacker, our encoding will require a large number of inherently sequential calls to some key derivation function KDF, which we model as a random oracle $\mathcal{O}$. We will then use these derived keys to encode the data.

A naïve approach would be to encode each block of a file F separately, making d calls to $\mathcal{O}$ for each; to compute $G[i]$, we compute $\kappa_i = \mathcal{O}^d(i \,\|\, F[i])$ and then encode $F[i]$ using κ_i (e.g., by letting $G[i] = \kappa_i \oplus F[i]$ if $|\kappa_i| = |F[i]|$). To avoid a costly computation the attacker can store either the encoded block or the derived key. If both blocks and keys are λ bits, both options require at least $|G|$ storage, making the encoding incompressible. This approach, however, is extremely costly, requiring nd oracle calls for an n-block file.

We could instead begin with Pietrzak's construction [45] based on a PoS. As we discussed in Section 2.2, this construction forecloses the possibility of rapid decoding, which we consider critical. We do, however, follow the lead of of prior and concurrent work on both proofs of inherently sequential work [18, 42] and proofs of space [6, 25, 30, 45, 48] and define our encoding as a directed acyclic graph (DAG). In particular, many PoS constructions compute a *labeling* of a DAG where the label of each vertex is derived from the labels of its parents using a hash function or suitable KDF.

In this mold we first construct a DAG whose labeling is a tight PoS, but with extra structure. This extra structure allows us to weave meaningful data through the whole computation in a way that preserves the incompressibility while allowing efficient decoding. Specifically, we partition the edges in our graph into *key edges* and *data edges*. Data edges represent lossless flows of data; the operation performed on blocks passed along data edges must transform the data in an invertible fashion. As passing data through a random oracle is not invertible, we use a keyed pseudorandom permutation PRP_κ in this case. Key edges represent (lossy) dependencies used to determine keys, and thus how other data is transformed. As depicted in Figure 2, at

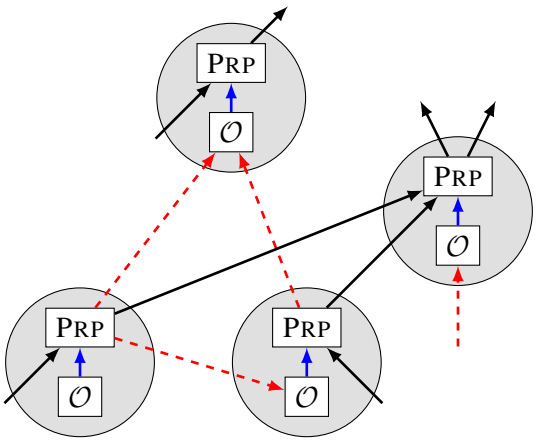


Figure 2: The high level structure of an encoding based on a DAG. Solid black edges are data edges, dashed red edges are key edges, and blue arrows from $\mathcal{O}$ to PRP show the use of derived keys.

each node we call $\mathcal{O}$ on all data passed in along key edges, and use the resulting key to permute data passed in along data edges using PRP_κ.

4.1 Encoder Graphs

The data encoding method on a DAG described above is not always invertible. Undoing the operation at a given vertex requires as input the values along all outgoing data edges and the values along all incoming key edges. A decoder graph representing these inverse dependencies can be defined by reversing the direction of appropriate edges. The encoding is invertible if the decoder graph is also a DAG. On the other hand, a cycle in the decoder graph would indicate some intermediate state that is necessary for decoding and yet not computable from encoding output.

We use this intuition to formally define *n-encoder graphs*.

Definition 2 (Encoder graphs)**.** Let $\mathcal{G} = (V, E)$ be a DAG where E can be partitioned into data edges E_D and key edges E_K. For a vertex $v \in V$, let $\text{DEG}^D(v)$ denote the number of incident edges in E_D and $\text{DEG}^K(v)$ the number of incident edges in E_K. We say that $\mathcal{G}$ is an *n-encoder graph* if the following hold.

1. $\mathcal{G}$ contains n source nodes and n sink nodes.
2. For all $v \in V$, either
   a) v is a source or sink node that produces or consumes a single block of data ($\text{DEG}^D(v) = 1$) and has no key edges in or out ($\text{DEG}^K(v) = 0$), or
   b) v produces the same amount of data as it consumes ($\text{DEG}^D_{\text{in}}(v) = \text{DEG}^D_{\text{out}}(v)$).
3. $\mathcal{G}$ represents an invertible transform. Specifically, there are no cycles in $\mathcal{G}^{-1} = (V, E^{-1})$, where

$$\begin{aligned} E^{-1} = &\{(u, v) \mid (v, u) \in E_D\} \\ &\cup \{(u, v) \mid \exists w \in V \text{ such that} \\ &\qquad (w, u) \in E_D \text{ and } (w, v) \in E_K\}. \end{aligned}$$

We say that $\mathcal{G}$ is *data a-regular* if $\text{DEG}^D_{\text{in}}(v) = a$ for all non-source, non-sink vertices v.

Construction 1. Let $\mathcal{G} = (V, E)$ be a data a-regular n-encoder graph where v_i denotes the ith vertex in a fixed topological sort. Given randomness ρ and input input $F = F[1] \,\|\, \cdots \,\|\, F[n]$ where $|F[i]| = \frac{\lambda}{a}$ for all i, we first compute $\kappa = \textsc{Kdf}(\rho \,\|\, F)$.

We assign a value to each edge in E as follows.

- For input vertices $v_1, \ldots, v_n$, assign $F[i]$ to v_i's single outgoing (data) edge.
- If v_i is neither a source nor sink, let $x_1, \ldots, x_k$ be the values assigned to incoming key edges and $y_1, \ldots, y_d$ be the values assigned to incoming data edges.

$$\kappa_i = \textsc{Kdf}(\kappa \,\|\, i \,\|\, x_1 \,\|\, \cdots \,\|\, x_k)$$
$$y' = y'_1 \,\|\, \cdots \,\|\, y'_d = \textsc{Prp}_{\kappa_i}(y_1 \,\|\, \cdots \,\|\, y_d).$$

  Output y' along each outgoing key edge. For the d outgoing data edges $e^i_1, \ldots, e^i_d$ ordered by the topological sort of their terminal vertices, output y'_i along e^i_i.

Let $y^\star_1, \ldots, y^\star_n$ be the values assigned to the incoming (data) edges of each sink node in $\mathcal{G}$. We define

$$\textsc{Encode}_{\mathcal{G}}(F\,;\,\rho) = (y^\star_1 \,\|\, \cdots \,\|\, y^\star_n, \kappa).$$

Condition 3 formalizes the intuition about decoding. To invert the computation at v, we need the values originally output along all data edges, so those edges reverse direction. We also need the values of each incoming key edge (w, v). The values produced by w during encoding are now produced by the vertices where w's outgoing data edges terminate, so we replace (w, v) with $\textsc{deg}^D_{\text{out}}(w)$ new edges. $\mathcal{G}$ represents an invertible transformation if this new $\mathcal{G}^{-1}$ has no cycles.

Note that Definition 2 is far from totally general. For example, it does not allow the encoding to condition on data or modify the file size at all.

4.2 Building an Encoding

Given a data a-regular n-encoder graph $\mathcal{G}$, we can define an encoding following the intuition of Figure 2. At each vertex v, we invoke our slow oracle $\mathcal{O}$ on all incoming key edges to derive a key κ_v. We then permute the concatenation of all incoming data edges using a pseudorandom permutation (PRP) keyed by κ_v. The result is sent out along all outgoing key edges and partitioned among outgoing data edges.

To formalize these, we let $\textsc{Kdf} : \{0,1\}^* \to \{0,1\}^\lambda$ be a key derivation function that we model as a random oracle, and $\textsc{Prp}_\kappa : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a keyed PRP that we model as an ideal cipher. We assume for simplicity that a evenly divides λ. Using this notation, Construction 1 specifies how to construct an efficiently invertible encoding.

Note that if $\mathcal{G}$ is not data a-regular, it is still possible to construct an efficiently invertible encoding using this technique. Doing so requires families of KDFs and PRPs to operate on each of the possible data sizes at each vertex.

4.3 Guaranteeing Sequential Work

Encoder graphs and Construction 1 define only an invertible encoding, they say nothing about its incompressibility. In order to construct a PIE, the graph $\mathcal{G}$ needs to ensure that a adversary that discards data will

need to make a large number of inherently sequential calls to KDF to recompute that data. As the edges in $\mathcal{G}$ represent data dependencies, paths in $\mathcal{G}$ represent inherently sequential sequences of operations. This means that $\mathcal{G}$ must contain long paths to ensure significant sequential work.

The path lengths in $\mathcal{G}$, however, determine only the work needed for the initial encoding. The security of a PIE requires that no adversary $\mathcal{A}$ can recompute discarded data efficiently, even if it is storing *intermediate* data. We can use the structure of $\mathcal{G}$ to analyze how $\mathcal{A}$ can use this intermediate data and ensure that there are still long paths of computational dependencies.

Depth-robust graphs. To address essentially the same concern, much prior and concurrent work [25, 30, 45] relies on *depth-robust graphs* (DRGs). Originally due to Erdös, Graham, and Szemerédi [26], a DRG is a graph that retains high depth even when many vertices are removed.

Definition 3 (Depth Robustness)**.** A graph $\mathcal{G} = (V, E)$ is (α, β)-*depth-robust* if for all $\tilde{V} \subseteq V$ with $|\tilde{V}| \geq \alpha|V|$, the induced subgraph on $\tilde{V}$ contains a path of length at least $\beta|V|$.

Prior work uses DRGs not only for proofs of space, but also to construct memory-hard functions [2–4, 12] and ensure inherently sequential work [42]—a use very similar to both ours and that of proofs of space.

We can straightforwardly construct an n-encoder graph from a DRG $\mathcal{G}$ by converting all edges in $\mathcal{G}$ into key edges and adding source an sink nodes for each existing vertex connected via data edges. The resulting encoding, however, is not a PIE. Since $\mathcal{G}$ is a DAG, it must contain at least one source node, and an adversary $\mathcal{A}$ can simply discard the output produced by that vertex and recompute it very rapidly if queried. Pietrzak [45] showed how to overcome this attack in to construct a tight proof of space by only considering the outputs from a subset of the vertices. When applied as a PIE, this technique would unfortunately require an honest decoder to recompute any part of $\mathcal{G}$ not used as output, again making decoding an inherently sequential operation.

Multiple DRGs. To retain the ability to decode efficiently while attempting to mitigate the above attack, we can attempt to layer multiple copies of $\mathcal{G}$. We can connect each vertex of one copy with a unique vertex of the next, and again make all edges in all copies of $\mathcal{G}$ key edges. This construction still leaves an attack: if $\mathcal{A}$ discards an ϵ fraction of blocks, we can guarantee that a long path exists in at least one copy of $\mathcal{G}$, but the end of that path may no longer connect to any output vertex of the encoder graph. Thus $\mathcal{A}$ may not need to recompute any such values.

To ensure security, we must guarantee that the end of any such long path connects to an output of the encoder graph. The easiest way to do this is to require that every input to one DRG layer depend on every output from the previous. This way, if there is a long path of key edges in one layer, the end of that path will connect to some vertex in the next layer, which will connect to some vertex in the layer above that, and so on, until we reach an output vertex.

We must, however, take care performing this connection. If the connection between two layers is brittle, $\mathcal{A}$ can sever it with a small amount of intermediate data. For example, multiple passes of a block cipher in CBC mode—originally suggested as an entire encoding scheme by Filecoin [9]—appears to create the required connection. Unfortunately, as discussed in Section 2.2, any reasonable number of passes is susceptible to "shortcut" attacks.

4.4 Dagwood Sandwich Graphs

There is a much more robust connector than a series of CBC passes, namely a *superconcentrator* [56].

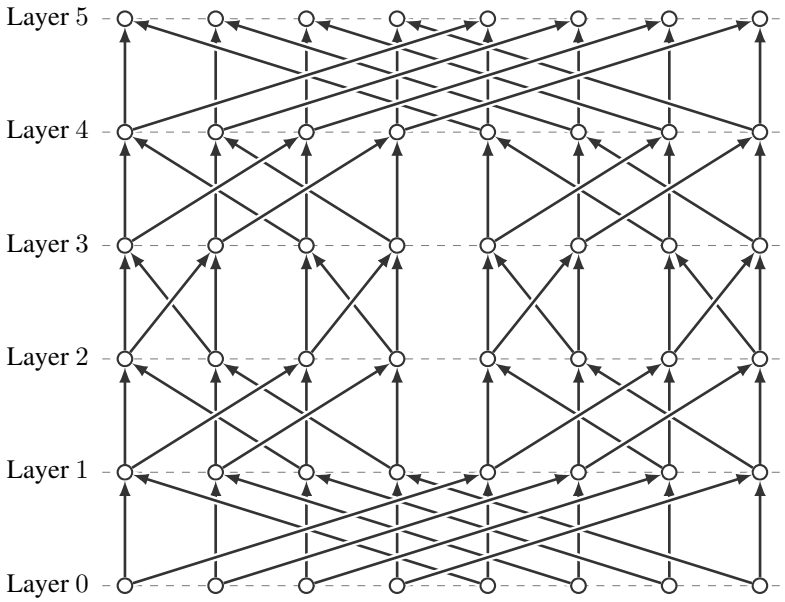


Figure 3: $\mathcal{B}_3$, a butterfly-based superconcentrator on 8 elements.

Construction 2 (Dagwood Sandwich Graph)**.** Let $\mathcal{G}$ be a (α, β)-DRG with $n = 2^k$ vertices, $\mathcal{B}_k$ by the butterfly n-superconcentrator, and $\ell = \left\lceil \frac{1}{1-\alpha} \right\rceil$. We construct a *Dagwood Sandwich Graph* (DSaG) $\widehat{\mathcal{G}}$ as follows.

Let $\mathcal{G}^1, \ldots, \mathcal{G}^\ell$ be ℓ independent copies of $\mathcal{G}$ and $\mathcal{B}^1, \ldots, \mathcal{B}^{\ell-1}$ be $\ell - 1$ independent copies of $\mathcal{B}_k$. Connect each vertex of $\mathcal{G}^1$ to two new vertices that are inputs of $\widehat{\mathcal{G}}$. Similarly, connect each output of $\mathcal{G}^\ell$ to two new output vertices of $\widehat{\mathcal{G}}$. For $i \in [1, \ell]$, connect the ith vertex of $\mathcal{G}^i$ to the ith vertex of $\mathcal{B}^i$ *twice*, and for $i < \ell$ connect the ith vertex of $\mathcal{B}^i$ to the ith vertex of $\mathcal{G}^{i+1}$, also twice.

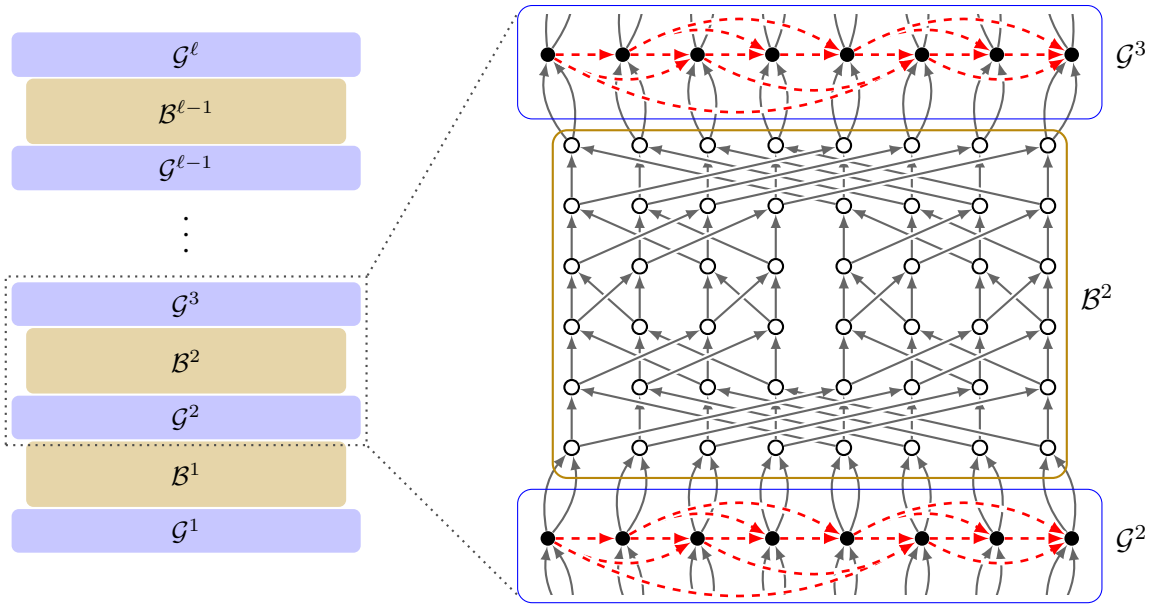
The vertices $\widehat{V}$ of $\widehat{\mathcal{G}}$ are all vertices specified above. The key edges $\widehat{E}_K$ are the edges of the $\mathcal{G}^i$s, and the data edges $\widehat{E}_D$ are all other edges.

Definition 4 (Superconcentrator)**.** Let $\mathcal{G} = (V, E)$ be a DAG with source vertices I and sink vertices O. We say $\mathcal{G}$ is an n-*superconcentrator* if $|I| = |O| = n$ and for all $r \in [0, n]$ and all $I' \subseteq I$ and $O' \subseteq O$ with $|I'| = |O'| = r$, there exist r vertex-disjoint paths connecting I' to O'.

While superconcentrators of linear size exist [33, 56], perhaps the simplest superconcentrator is the *butterfly superconcentrator*, created by connecting two classic butterfly graphs. Butterfly superconcentrators have the added benefit of being 2-regular—all (non-source/sink) vertices have both in and out-degree 2. As the number of inputs and outputs of such a graph must be a power of 2, we let $\mathcal{B}_k$ represent that butterfly 2^k-superconcentrator. Figure 3 shows $\mathcal{B}_3$.

Indeed, for a (α, β)-DRG $\mathcal{G}$ with n vertices, it is sufficient to layer $\ell = \left\lceil \frac{1}{1-\alpha} \right\rceil$ copies of $\mathcal{G}$, each connected by an n-superconcentrator. We refer to this multi-layered construction as a *Dagwood Sandwich Graph* (DSaG), shown graphically in Figure 4. We define a DSaG formally in Construction 2.

With $\left\lceil \frac{1}{1-\alpha} \right\rceil$ layers, no adversary $\mathcal{A}$ can eliminate all long paths in all copies of $\mathcal{G}$ with less data than the full file size. Moreover, the vertex-disjointness condition in Definition 4 prevents $\mathcal{A}$ from gaining any advantage by storing data inside these connectors instead of DRGs. This means that Construction 2 defines a



(a) DSaG macro-structure

(b) Micro-structure of DSaG layers

Figure 4: The structure of a Dagwood Sandwich Graph. The left shows many layers of alternating DRGs ($\mathcal{G}^i$) and superconcentrators ($\mathcal{B}^i$), while the right shows the structure of the edges, with data edges in solid gray and key edges in dashed red.

data 2-regular $2n$-encoder graph that in turn defines a PIE using Construction 1.

Theorem 1. *If $\mathcal{G}$ is a (α, β)-DRG with n vertices, then* $\text{ENCODE}_{\widehat{\mathcal{G}}}$ *defined by Construction 1 on the DSaG $\widehat{\mathcal{G}}$ defined by Construction 2 is βn-oracle incompressible.*

We provide a proof of Theorem 1 in Appendix B.

5 Rapid Decoding and Practical Implications

We now explain how a DSaG-based PIE supports efficient data recovery. This feature is a key contribution that sets our PIE construction apart from prior approaches relying on existing proofs of space. It is also critical for a DSN wishing to function as a public CDN or similar application where users need to rapidly access data. Unfortunately, it results in some practical ramifications that we discuss below.

5.1 Rapid Decoding

Our construction provides two routes to rapid decoding. The first is parallelism, which applies generally regardless of the slow function used. The second replaces the slow KDF operation with an asymmetric-speed permutation, and allows even single-threaded decoding to far outpace encoding.

Parallel Decoding. We enforce inherently sequential work in ENCODE by making key dependencies within individual layers of a DSaG form a DRG. This means the output of encoding some vertices are necessary to compute the keys used to encode (and decode) other vertices *in the same layer*. The data values output by a vertex during encoding are, however, the values *input* to that vertex during decoding. This means that, in

order to derive derive the key necessary to *decode* a block in a DRG layer, we need only the *inputs* of other vertices in the same layer. Since these inputs are produced by the superconcentrator connecting DRG layers, they can easily be made available in parallel. Therefore, while decoding, we can perform every KDF call on a given DRG layer at the same time, massively reducing the time needed to decode a file.

Faster Sequential Decoding. Given as a primitive an asymmetric-speed PRP, we can significantly hasten decoding. Specifically, instead of modeling KDF as a slow random oracle and PRP_k as an ideal cipher, we can model KDF as a *fast* random oracle, and PRP_k as a cipher that is moderately slow to compute by highly efficient to invert. This results in a PIE with the same security, but much faster decoding.

Asymmetric PRPs are not new. For example, Lenstra and Wesolowski propose the Sloth permutation [40] based on modular square roots interleaved with a fast PRPs. Boneh et al. [11] note that separately applying such a PRP to each λ-sized file block is sufficient to construct a PIE without a complicated encoding scheme. A DSaG complements that approach by massively increasing the sequential work needed to encode a file, while still allowing for rapid decoding.

As we show in Section 8.3, both of these approaches are borne out in practice. Indeed, both allow our DECODE operation to complete in significantly less time than even the inherently-sequential security bound placed on ENCODE.

5.2 Practical Implications

Rapid decoding appears to effectively support settings like a public CDN, but it is not without drawbacks.

Composition. Careful readers will note that Definition 1 for incompressibility is only secure under composition—parallel or sequential—if the choice of files in one execution of $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}$ is independent from all other executions. When the files *are* independent, we can simply combine multiple games into a single game with the union of the file sets. Without this guarantee, however, composability is impossible.

Consider composing two runs with one file each. In the first run F_1 is an arbitrary file with correct encoding G_1 with randomness ρ_1. In the second, we let $F_2 = G_1$ and let $G_2 = \text{ENCODE}(G_1\,;\,\rho_2)$. Since $\mathcal{A}$ can *decode* rapidly, it can then store only G_2 and easily recover G_1 on-the-fly. *We stress that this lack of composition is inherent to any PIE or proof of space with public randomness and efficient data extraction [45], it is not a property of our PIE scheme in particular.*

Limitations for DSNs. The above lack of composition means that any DSN must somehow ensure that files are generated independently of the encoding. A natural way to do this is to force storage providers to commit to their files prior to selecting the randomness—thus reducing to a single execution of $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}$. This imposes very little overhead if the files are stating and encoded all at the same time. It unfortunately renders it nearly impossible to add or modify files later. Because additions or modifications could be conditioned on already-encoded files, we must re-encode *all* files for *any* addition or modification. With no other general mechanism for showing generic files are independent, this appears to be an inherent limitation.

In setting such as Filecoin which must both allow new data to enter the system and account for adversarial files, the DSN must operate in epochs and re-encode every file for each epoch. As we discuss in Section 7, our construction provides a nearly optimal amount of computation for a given security bound, allowing us to estimate the financial cost of such a mechanism. Taking the performance results from Section 8.2, using a Sloth-based PIE to encode large file chunks (over 1 MB) requires around 14.5 sec/MB of CPU time on a `c5.xlarge` Amazon EC2 instance.[3] Even using spot instances—currently priced at 1.9¢/hour [5]—this would cost an honest provider 7.8¢/GB for each encoding. Assuming epochs of at most one day so that data

[3]This is the speed of the fastest Sloth setting. Larger file chunks would thus increase the minimum sequential work, but not reduce the total time.

can be modified or added frequently, this price balloons to a monthly cost of $2.34/GB simply for re-encoding. Note that this is the cost for one replica; triplicate replication would impose 3× overhead. Moreover, this only accounts for computational overhead; it does not consider the costs of accessing data, which may be substantial. This raises serious questions about the cost-effectiveness for anything other than high-value data.

DSNs for Specialized Files. We note, however, that these restrictions—and their costs—do not apply when data is known to be independently or "honestly" generated. This means our scheme supports highly efficient public DSNs for restricted classes of files. For example, blockchain state data is by assumption honest. Ethereum could use a PIE to prove redundant storage of critical blockchain data, enhancing, for instance, their proposed "availability proof" scheme [1]. As Ethereum has averaged adding over 200 MB of state per day for the last year [27], this could provide significant benefit.

6 Optimization

There are two different ways to optimize this construction without compromising its security. The first removes calls to KDF when they are not necessary for security, and the second allows for increased parallelism during encoding.

6.1 Removing Unnecessary Slow Operations

The proof that a DSaG produces a PIE relies on the fact that paths with d key edges require d inherently sequential calls to KDF to compute. To ensure this, we need any vertex with an incoming key edge to call KDF and supply that key data (and possibly more) as an argument. Construction 1 does this by calling KDF at *every* vertex.

Vertices with no incoming key edges, however, need not call KDF to ensure this property. In fact, an adversary needs no intermediate state to recompute such keys, meaning the work cannot be sequential, so the slowness of these KDF calls provides no security. Instead of KDF, we can call some other, faster key derivation function KDF$'$ at these vertices. We still model KDF$'$ as a random oracle $\mathcal{O}'$, but we need not bound the number of calls to $\mathcal{O}'$. This means that, in practice, we can implement $\mathcal{O}'$ using a fast function, like a SHA hash, while we still need a slow hash function for $\mathcal{O}$.

If $\mathcal{G}$ has many vertices with no incoming key edges, this can significantly reduce the runtime of ENCODE$_{\mathcal{G}}$ without impacting security. As our DSaG construction in Section 4.4 contains *mostly* vertices with no incoming key edges, this optimization makes a dramatic impact.

6.2 Chunking Files

Meaningful parallelization during encoding appears impossible by construction. We designed ENCODE to require inherently sequential work. For large files, however, the sequential work guaranteed by a single DSaG may be far larger than necessary to enforce practical security. Moreover, the DSaG itself may scale poorly to large file sizes, introducing unnecessary overhead in such cases.

To bypass these impediments, we note that multiple disconnected copies of a DSaG $\widehat{\mathcal{G}}$ together have all of the properties necessary to produce a PIE. This stems from the fact that m disconnected copies of an (α, β)-DRG together form a $(\alpha, \beta/m)$-DRG. As the copies are disconnected, we do not need to connect them to each other using superconcentrators and our security arguments will still hold. Specifically, m copies of an n-input DSaG $\widehat{\mathcal{G}}$ built using a (α, β)-DRG will produce a βn-oracle incompressible encoding on mn blocks.

Because the copies of $\widehat{\mathcal{G}}$ are disconnected, we can split the file into n-block chunks and encode those chunks in parallel. This insight provides the additional benefit that we only need a single graph $\widehat{\mathcal{G}}$, and the chunk size presents an opportunity for configuration. With small chunks more parallelism is possible and padding out files to an even number of chunks is inexpensive, allowing efficient use of the same graph for small or oddly-sized files. Large chunks mean larger βn, requiring more inherently sequential work. This allows us to reduce the cost of a single KDF call while maintaining wall-clock timing assumptions, making (sequential) encoding faster.

Regardless of the parameterization, any fixed chunk size n allows the encoding time to scale efficiently with large file size. Specifically, the cost of encoding any F is now $O\left(\frac{|F|}{n} \cdot |\mathcal{G}|\right)$—linear in $|F|$.

7 Security Efficiency Ratio

Measuring the efficiency of a PIE is not as straightforward as measuring the performance of ENCODE and DECODE. ENCODE must be slow to provide security, but we do not want it to be slower than necessary. Instead, we compare the execution time of ENCODE to the guaranteed sequential work needed to recompute a discarded block. We call this ratio the *security efficiency ratio* (SER). An SER of 1 would indicate that the time needed for ENCODE is the same as the time needed to recompute a single discarded block—a bound no secure PIE can ever break. The SER therefore measures how far ENCODE's performance is from the theoretical optimum.

If a PIE allows parallelism during encoding, its SER will vary with parallelism. While it is possible to cut a file into chunks and parallelize encoding across chunks (Section 6.2), DSaGs preclude parallelism within a file chunk. We thus attempt to minimize the sequential SER of a single chunk.

Ideally the cost of ENCODE will be dominated by the number of calls to KDF. Using the fast KDF′ optimization from Section 6.1, we need one KDF operation for each vertex with an incoming key edge. For a DSaG, this is precisely the vertices in DRG layers. If the DRG is (α, β)-depth-robust with n vertices, the minimum sequential work requires βn calls to KDF, while the total work requires ℓn calls. Therefore the SER is at least $\ell n / \beta n = \ell / \beta$. Construction 2 sets $\ell = \left\lceil \frac{1}{1-\alpha} \right\rceil$, meaning

$$\text{SER} \geq \frac{1}{(1-\alpha)\beta}.$$

If both α and β are constants—which is often a goal of DRG construction—this ratio is independent of n.

To see what DRGs will produce the most efficient PIEs, we note that no graph can have depth-robustness better than (α, α). Thus we aim to minimize $\frac{1}{\alpha(1-\alpha)}$. Since $\alpha \in (0, 1)$, we achieve this minimum at $\alpha = \frac{1}{2}$, bounding the SER at 4.

This bound may be loose for two reasons: overhead from fast operations (e.g., KDF′) and nonuniform runtime of KDF. The first is straightforward and, as we see in Section 8.1, is minimal in our implementation. The second results from varying in-degree of the DRG. The in-degree of a vertex determines the input length of the corresponding KDF call, so longer inputs take more time. Since we cannot force recomputation of any specific vertices, our wall-clock security bound conservatively assumes the minimum input length for every KDF call. As this bound defines the "security" portion of SER, larger in-degree graphs may have higher SER for a given α and β.

8 Experiments

We now present performance results for our PIE implementation. Our PIE implementation is 650 lines of Java code that rely on BouncyCastle [13] for all hash and cipher primitives. Except where otherwise noted, we ran all benchmarks on a `c5.xlarge` Amazon EC2 instance, which provides 2 cores of an Intel Xeon Platinum 8124M CPU.

We benchmark two constructions. The first is a provably-secure PIE based on a naïvely-constructed $(0.5, 0.5)$-DRG with in-degree linear in the graph size, providing a near-optimal SER of 4 on small file sizes, but scaling poorly on individual files. In Section 6.2 we note that we can break large files into independent chunks to maintain good performance. Here we use only fast PRPs and instantiate a slow KDF as scrypt [44]—the popular memory-hard password hashing function with highly tunable performance. The second is a heuristically-secure PIE using a randomly-sampled graph specified by Fisch et al. [32] which we heuristically assume is $(0.5, 0.25)$-depth-robust based on Fisch's experimental results. This provides an SER bound of 8 and much better scaling. To enable extra-fast decoding, we use only fast KDFs and employ Sloth [40] as an asymmetric-speed PRP. We provide more detail on these constructions in Appendix C.

In both cases we instantiate all fast PRPs as Threefish [28]—a block cipher with a fast 512-bit implementation in BouncyCastle—and all fast KDFs as SHA512. We additionally use fast KDFs and PRPs in the superconcentrator layers of the encoding as discussed in Section 6.1.

8.1 Micro-benchmarks

Our security model assumes all operations except the slow KDF or PRP are instantaneous. To confirm that they add minimal overhead in practice, we benchmarked each operation separately. Indeed, Threefish on a single 512-bit block took only 0.36 μs, while SHA512 took 0.67 μs on small inputs.

We also benchmarked single iterations of both the butterfly superconcentrator and the DRG transformation within our PIE. For each transform, we measured the runtime for a variety of graph sizes. In the DRG layer, we note that SHA512's compression algorithm is far faster than scrypt's, so even when using scrypt we use SHA512 to first compress all key inputs. We therefore include SHA512's compression time, but not timing for either scrypt or Sloth. The results in Figure 5 show that all layers impose very small overhead for reasonably-sized graphs, but the naïve DRG scales poorly.

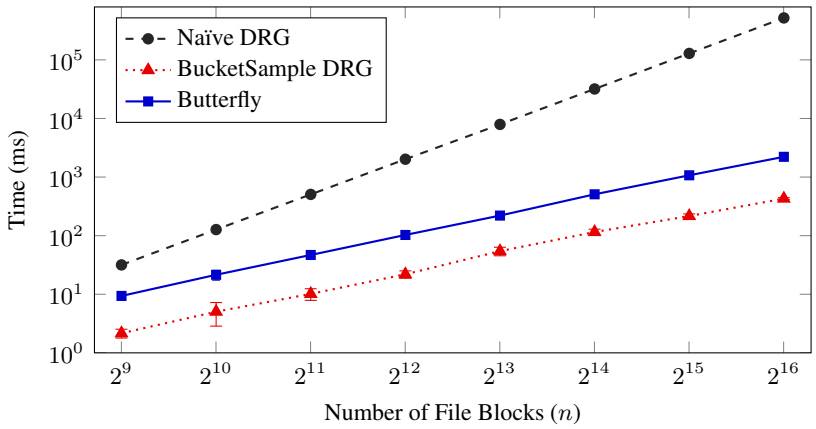


Figure 5: Fast operation overhead for different layer types and sizes. Naïve DRG run time grows quadratically with the graph size, while the others grow roughly linearly.

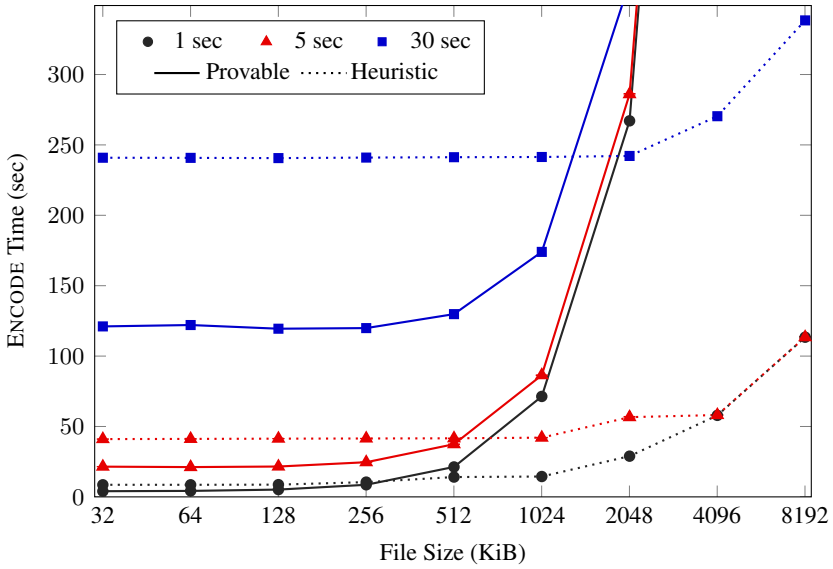


Figure 6: ENCODE time by file size for wall-clock security bounds of 1, 5, and 30 sec. Solid lines use scrypt as a slow KDF and a provably-secure naïve DRG that scales poorly to large files. Dotted lines use a heuristically depth-robust graph and Sloth as a slow PRP. Here performance slows because Sloth's fastest configuration takes 420 μs per iteration. Overhead of "fast" operations remains minimal.

8.2 ENCODE Performance

In order to ensure that the formal security of a PIE corresponds to practical security, we need our "slow" operation to be sufficiently slow. In particular, if an encoding is d-oracle incompressible, the time required for d sequential executions must be noticeable. By configuring the slow operation to different speeds, we can tune this value.

For both implementations we benchmarked performance for different speeds of the slow operation and different file sizes. We measured files ranging from 32 KiB to 4 MiB in size. Larger files can be broken into independent chunks and transformed in parallel, as described in Section 6.2. For each file size we tuned the inherent sequential work to require roughly 1, 5, and 30 sec on the machine running the encoding. Figure 6 presents the results of these benchmarks.

For small files, the provably-secure PIE with a naïve DRG performs extremely well. Even with a security bound of 1 sec, the SER ranges from 4.1 to 5.0 for files up to 128 KiB. For larger security bounds, scrypt calls dominate the runtime longer, with the SER between 4.0 and 4.6 for files up to 0.5 MiB with a 30-sec bound. Unfortunately, the poor scaling of the naïve DRG completely dominates the encoding time of 2 MiB files, resulting in a SER over 13 with a 30-sec security bound and over 100 with a 1-sec bound.

The heuristically-secure PIE, however, scales extremely well. While encoding time increases for larger file sizes, this because Sloth's speed cannot be tuned to a fine enough granularity. The only tunable parameter is the number of modular square roots per invocation, and a single square root takes around 420 μs on a `c5.xlarge`. The minimum sequential work for a 4 MiB file is 2^{14} Sloth invocations, which takes at least 6.8 seconds. This explains the convergence of the 1 and 5-second bounds. Indeed, in every heuristically-secure experiment the measured SER remained between 8.0—the minimum for this construction—and 8.5.

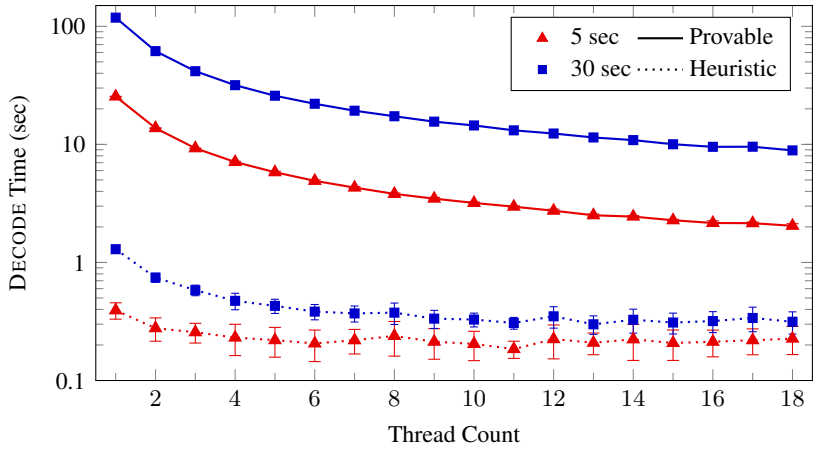


Figure 7: DECODE time for a 256 KiB file for both 5 and 30-sec security bounds run with multiple threads on an Amazon EC2 `c5.18xlarge`. Solid lines use a provably-secure naïve DRG and scrypt. Dotted lines use a heuristic DRG and Sloth.

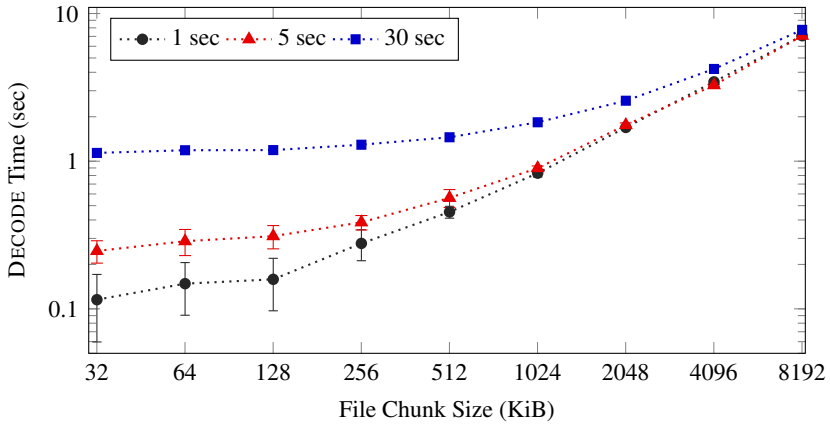


Figure 8: DECODE time by file size for 1, 5, and 30-sec security bounds using a heuristically-secure DRG and the Sloth PRP.

8.3 DECODE Performance

In Section 5.1 we explain how to achieve rapid decoding through either parallelism or asymmetric-speed PRPs. We now benchmark the performance of both of these techniques.

We benchmarked DECODE with various numbers of threads on a single 256 KiB file for both a naïve DRG using scrypt and a heuristic DRG using Sloth. To enable high parallelism, we ran this benchmark on a `c5.18xlarge` Amazon EC2 instance, which provides 72 virtual CPU cores. We benchmarked configurations requiring inherently-sequential *encoding* work of both 5 and 30 sec for each setup. The results in Figure 7 show that parallelism is highly effective at improving DECODE performance when using scrypt. With Sloth, decoding is fast enough with a single thread that parallelism provides little benefit. In fact, for Sloth with a 5-sec security bound, increasing the thread count provides no noticeable benefit beyond 5 threads.

Finally, we benchmarked single-threaded DECODE performance when using Sloth for a variety of different file sizes. As shown in Figure 8, the performance depends primarily on the inherently sequential

ENCODE time for very small files, as a large number of Sloth decode operations is slower than the other fast operations. As the file size grows, however, the tuning of Sloth gets increasingly small, meaning overhead from the larger graphs becomes noticeable and the running time begins to scale linearly with graph size.

9 Related Work

Proofs of Storage. As noted above, PoRs [23, 38, 53] and PDPs [7], often generically called proofs of storage [39], prove only file retention by a provider, not file replication. As PoRs involve erasure coding, they may be viewed as proofs of replication in the trusted-encoder, private-reader setting. Multi-replica PDPs, e.g., [19], may be viewed similarly.

Proofs of space, introduced by Dziembowski et al. [25] as an alternative to proof of work in blockchains, are conceptually related. They help ensure that a prover is consuming a certain amount of storage. Proofs of space, however, involve specially constructed files, not generic files as in a DSN.

Proofs of Replication. Several early-stage DSNs, such as Sia [58] and Storj [59], already perform proofs of replication in the trusted-encoder, private-reader setting.

An encoding technique not intended for but useable for proving replication in the trusted-encoder, public-reader setting was proposed by van Dijk et al. [57] based on the RSA trapdoor one-way permutation. Damgård, Ganesh, and Orlandi [21] propose a similar construction, but embellish it using PoRs to support file extraction, rather than just incompressibility, with in-depth analysis focused on file replication.

Previous attempts to construct PIEs [9, 47, 57] were noted above. Fisch et al. [31] first proposed use of DRGs to address the untrusted-encoder, public-reader file replication problem, a stepping stone towards our use of DRGs in this work. Fisch independently and concurrently proposed a security notion for proof-of-replication called ϵ-rational replication that directly addresses the rational behavior of an adversary and shows that PIEs are both necessary and sufficient to realize this definition [29]. Fisch also constructed an alternative PIE using bipartite expander graphs instead of superconcentrators [30].

Depth-Robust Graphs. DRGs were studied decades ago for proving lower bounds on computational complexity [26, 51, 52]. DRGs have since proven useful for enforcing sequential work in publicly verifiable proofs of work [42] and memory-hard hash functions [2–4, 12], prompting a revival of interest as well as new DRG constructions [2, 42].

Incompressible Functions. Various notions of incompressibility used in cryptographic security proofs, e.g., [37], and protect keys for digital-rights management [24], do not relate directly to our setting. More relevant are storage-enforcing commitments [36], which bound compressibility to prove that a certain, minimal amount of storage is devoted to a file F. They do not enable proof of replication, though.

Publicly Verifiable Proofs of Sequential Work and Delay Functions. Publicly verifiable proofs of sequential work, including those of Cohen and Pietrzak [18] and Mahmoody et al. [42], which uses DRGs, are public-coin systems that enable a prover to show that it has done a certain amount of sequential work relating to a particular input. They enable fast verification (polylog in the time of the prover), which our construction does not, but are intended for applications that do not involve data recovery (e.g., timestamping, CPU benchmarks). As a result, they do not support decoding, which is needed for proofs of replication.

Delay functions, introduced by Goldschlag and Stubblebine [35], are similar, but do enforce uniquely computable outputs and can be decodable. Boneh et al. [11] show how to construct verifiable delay functions (VDFs) that are decodable and, in principle, usable for proofs of replication. They show how to make verification fast (again, polylog in the prover's time) using a SNARK framework called incremental verifiable computation—something not possible in our DSaG construction. Their constructions are theoretical, though,

with no reported implementation. The proposed VDFs involve chained operations on individual (e.g., 256-bit) field elements composing a file, so their SERs are impractically high (e.g., SER $\approx$ 8,000 for a 128 KiB file).

10 Conclusion

We have presented the first practical, provably secure, and efficiently decodable *public incompressible encoding* (PIE), a tool for proofs of replication in the public setting. PIEs enable detection of servers that fail to use adequate storage, thus causing economically rational adversaries to correctly store fully redundant data. We have formally defined PIE security and show how to construct efficient, provably secure PIEs using *Dagwood Sandwich Graphs* (DSaGs), an iterated interleaving of depth-robust graphs (DRGs) and superconcentrators.

A number of interesting problems arise from our work. There is the challenge of efficient verification of correct PIE computation while retaining fast data extraction and avoiding impractically heavyweight proofs systems. There is also the problem of supporting *dynamic* provable replication: replicating files whose contents change over time. This is trivially achievable by updating modified file chunks, but impractical for sparsely distributed file changes, as the inefficiency of existing approaches suggests [45].

In summary, we believe that the PIE constructions we present here—allowing fast decoding and providing near-optimal efficiency in terms of SER—provide powerful insights into where secure public-setting DSNs may or may not be practical. They are a promising practical tool for some application domains, such as robust blockchain data storage, while raising questions about the cost-efficiency of others.

Acknowledgements

Many people helped with this work. Nicola Greco and Juan Benet at Protocol Labs helped structure the problem and clarify practical requirements, and their concurrent work involving Joe Bonneau pushed us toward strong adversarial models. Mic Bowen at Intel and Samarth Kulshreshtha pointed us to new potential applications of PIEs. Finally, Lorenz Breidenbach, Rahul Chatterjee, and Paul Grubbs asked many insightful questions and helped with editing.

Funding for this work was provided by a National Defense Science and Engineering Graduate Fellowship, NSF grants CNS-1330599, CNS-1514163, CNS-1564102, and CNS-1704615, as well as ARO grant W911NF-16-1-0145 and IC3 industry partners. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect those of these sponsors.

References

[1] M. Al-Bassam, A. Sonnino, and V. Buterin. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. `https://arxiv.org/abs/1809.09044`, 2018.

[2] J. Alwen, J. Blocki, and B. Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In *Conference on Computer and Communications Security (CCS)*, pages 1001–1017. ACM, 2017.

[3] J. Alwen, J. Blocki, and K. Pietrzak. Depth-robust graphs and their cumulative memory complexity. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 3–32. Springer, 2017.

[4] J. Alwen, P. Gazi, C. Kamath, K. Klein, G. Osang, K. Pietrzak, L. Reyzin, M. Rolínek, and M. Rybár. On the memory-hardness of data-independent password-hashing functions. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 51–65. ACM, 2018.

[5] Amazon EC2 pricing. `https://aws.amazon.com/ec2/pricing/`. Accessed November 2018.

[6] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks (SCN)*, pages 538–557. Springer, 2014.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Conference on Computer and Communications Security (CCS)*, pages 598–609. ACM, 2007.

[8] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *Conference on Computer and Communications Security (CCS)*, pages 863–874. ACM, 2013.

[9] J. Benet, D. Dalrymple, and N. Greco. Proof of replication. Technical report, Protocol Labs, July 27, 2017. `https://filecoin.io/proof-of-replication.pdf`. Accessed June 2018.

[10] K. Benson, R. Dowsley, and H. Shacham. Do you know where your cloud files are? In *Cloud Computing Security Workshop (CCSW)*, pages 73–82. ACM, 2011.

[11] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. *IACR Cryptology ePrint Archive*, June 2018. `https://ia.cr/2018/601`.

[12] D. Boneh, H. Corrigan-Gibbs, and S. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *Advances in Cryptology (ASIACRYPT)*, pages 220–248. Springer, 2016.

[13] Bouncy Castle Crypto APIs (Version 1.59). `https://www.bouncycastle.org/`, Dec. 28, 2017.

[14] K. D. Bowers, M. Van Dijk, A. Juels, A. Oprea, and R. L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Conference on Computer and Communications Security (CCS)*, pages 501–514. ACM, 2011.

[15] V. Buterin. The stateless client concept. Oct. 24, 2017. `https://ethresear.ch/t/the-stateless-client-concept/172`. Accessed June 2018.

[16] R. Canetti, B. Riva, and G. N. Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

[17] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In *EUROCRYPT*, pages 451–467, 2018.

[18] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 451–467. Springer, 2018.

[19] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 411–420. IEEE, 2008.

[20] W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*, chapter 4. 1 edition, 2004.

[21] I. Damgård, C. Ganesh, and C. Orlandi. Proofs of replicated storage without timing assumptions. *IACR Cryptology ePrint Archive*, July 2018. https://ia.cr/2018/654.

[22] Data storage supply and demand worldwide, from 2009 to 2020 (in exabytes). https://www.statista.com/statistics/751749/worldwide-data-storage-capacity-and-demand/. Accessed June 2018.

[23] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference (TCC)*, pages 109–127. Springer, 2009.

[24] C. Dwork, J. Lotspiech, and M. Naor. Digital signets: Self-enforcing protection of digital information (preliminary version). In *Symposium on the Theory of Computing (STOC)*, pages 489–498. ACM, 1996.

[25] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In *International Cryptology Conference (CRYPTO)*, pages 585–605. Springer, 2015.

[26] P. Erdös, R. L. Graham, and E. Szemeredi. On sparse graphs with dense long paths. Technical report, Stanford University, 1975.

[27] Etherscan.io. Ethereum ChainData Size. https://etherscan.io/chart2/chaindatasizefast. Accessed November 2018.

[28] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3, 2010.

[29] B. Fisch. Poreps: Proofs of space on useful data. *IACR Cryptology ePrint Archive*, July 2018. https://ia.cr/2018/678.

[30] B. Fisch. Tight proofs of space and replication. *IACR Cryptology ePrint Archive*, Aug. 2018. https://ia.cr/2018/702.

[31] B. Fisch, J. Bonneau, J. Benet, and N. Greco. Proof of replication using depth robust graphs. Talk at Blockchain Protocol Analysis and Security Engineering (BPASE) conference, 2018.

[32] B. Fisch, J. Bonneau, N. Greco, and J. Benet. Scaling proof-of-replication for filecoin mining. Technical report, Stanford University, 2018. https://web.stanford.edu/~bfisch/porep_short.pdf. Accessed November 2018.

[33] O. Gabber and Z. Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420, 1981.

[34] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *International Cryptology Conference (CRYPTO)*, pages 465–482. Springer, 2010.

[35] D. M. Goldschlag and S. G. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In *Financial Cryptography and Data Security (FC)*, pages 214–226. Springer, 1998.

[36] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography and Data Security (FC)*, pages 120–135. Springer, 2002.

[37] S. Halevi, S. Myers, and C. Rackoff. On seed-incompressible functions. In *Theory of Cryptography Conference (TCC)*, pages 19–36. Springer, 2008.

[38] A. Juels and B. S. Kaliski Jr. PORs: Proofs of retrievability for large files. In *Conference on Computer and Communications Security (CCS)*, pages 584–597. ACM, 2007.

[39] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security (FC)*, pages 136–149. Springer, 2010.

[40] A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptology ePrint Archive*, Apr. 2015. `https://ia.cr/2015/366`.

[41] S. D. Lerner. Proof of unique blockchain storage, 2014. `https://bitslog.wordpress.com/2014/11/03/proof-of-local-blockchain-storage/`. Accessed November 2018.

[42] M. Mahmoody, T. Moran, and S. Vadhan. Publicly verifiable proofs of sequential work. In *Innovations in Theoretical Computer Science (ITCS)*, pages 373–388. ACM, 2013.

[43] R. Pass and E. Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[44] C. Percival and S. Josefsson. The scrypt password-based key derivation function. Technical report, Aug. 2016.

[45] K. Pietrzak. Proofs of catalytic space. *IACR Cryptology ePrint Archive*, Feb. 2018. `https://ia.cr/2018/194`.

[46] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

[47] Protocol Labs. Filecoin: A decentralized storage network. Jan. 2, 2018. `https://filecoin.io/filecoin.pdf`. Accessed June 2018.

[48] L. Ren and S. Devadas. Proof of space from stacked expanders. In *Theory of Cryptography Conference (TCC)*, pages 262–285. Springer, 2016.

[49] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[50] L. Rizzatti. Digital data storage is undergoing mind-boggling growth. *EE Times*, Sept. 14, 2016.

[51] G. Schnitger. A family of graphs with expensive depth-reduction. *Theoretical Computer Science*, 18(1):89–93, 1982.

[52] G. Schnitger. On depth-reduction and grates. In *Foundations of Computer Science (FOCS)*, pages 323–328. IEEE, 1983.

[53] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology (ASIACRYPT)*, pages 90–107. Springer, 2008.

[54] Y. Sompolinsky and A. Zohar. Bitcoin's underlying incentives. *Communications of the ACM*, 61(3):46–53, 2018.

[55] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains. 2017. https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf.

[56] L. G. Valiant. On non-linear lower bounds in computational complexity. In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 45–53. ACM, 1975.

[57] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *Conference on Computer and Communications Security (CCS)*, pages 265–280. ACM, 2012.

[58] D. Vorick and L. Champine. Sia: Simple decentralized storage. https://sia.tech, 29 Nov. 2014. Accessed June 2018.

[59] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, C. Pollard, and V. Buterin. Storj: A peer-to-peer cloud storage network. https://storj.io/storj.pdf, 2016. Accessed June 2018.

Appendix

A Multi-Query Security

To ensure no adversary can gain advantage by amortizing computation across multiple queries, we provide a notion of incompressibility that queries $\mathcal{A}$ on multiple data blocks. This definition of is similar to Definition 1, but with multiple challenges. These challenges correspond to a real-world verifier querying a storage server on multiple data blocks in sequence, We therefore allow $\mathcal{A}$ to modify its data storage between each query so long as $\mathcal{A}$ never increases its total storage and can compute each piece of stored data from the previous within the query's time bound.

Definition 5 (Multi-Query Public Incompressible Encoding)**.** We say a public encoding algorithm ENCODE is *multi-query d-oracle incompressible* if for any set of files $\{F_i\}_{i=1}^m$, any compression factor ϵ, any number of queries s, and any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $negl$ such that

$$\Pr\left[\text{EXP}^{\text{MPIE}}_{\mathcal{A},\mathcal{O}}(\lambda, \{F_i\}, \epsilon, d, s) = 1\right] \leq (1-\epsilon)^s + negl(\lambda)$$

where

$\underline{\text{EXP}^{\text{MPIE}}_{\mathcal{A},\mathcal{O}}(\lambda, \{F_i\}, \epsilon, d, s)}$

$\{\rho_i\}_{i=1}^m \leftarrow\!\!\$\ \{0,1\}^{\lambda \times m}$

$G'_0 \leftarrow\!\!\$\ \mathcal{A}_1^{\mathcal{O}}(\{(F_i \rho_i)\}_{i=1}^m)$

for $i \in [1,m] : (G_i, \kappa_i) \leftarrow \text{ENCODE}^{\mathcal{O}}(F_i\,;\,\rho_i)$

$G = G_1 \,\|\, \cdots \,\|\, G_m$

for $j \in [1,s]$:

  $k_j \leftarrow\!\!\$\ [1, |G|]$

  $(blk_j, G'_j) \leftarrow\!\!\$\ \mathcal{A}_2^{\mathcal{O}^\star_d}(k_j, G'_{j-1}, \{(\rho_i, \kappa_i)\}_{i=1}^m)$

endfor

return $\bigwedge_{i=1}^{s} \left[\left(\frac{|G'_{i-1}|}{|G|} \leq 1-\epsilon\right) \wedge (blk_j = G[k_j])\right]$

This security property is conveniently equivalent to the single-query notion provided in Definition 1.

Proposition 1. *A public encoding algorithm* ENCODE *is d-oracle incompressible if and only if it is multi-query d-oracle incompressible.*

Proof. Multi-query incompressibility clearly implies single-query. We now consider the other direction. We will prove this by induction on s.

If $s = 1$, this is exactly the single-query game.

We now assume that an encoding scheme is multi-query public incompressible with s queries whenever it is single-query incompressible. We claim that the same holds for $s + 1$ queries. Let ENCODE be a (single-query) d-oracle incompressible encoding, and assume for the sake of contradiction that $\mathcal{A}$ breaks its multi-query security with $s + 1$ queries. Since ENCODE is multi-query incompressible on s queries, we know that the probability that $blk_j = G[k_j]$ for all $j \in [1, s]$ is at most $(1 - \epsilon)^s + negl(\lambda)$. However, by assumption, there is some non-negligible ν such that the probability of success on all $s + 1$ queries is greater than $(1 - \epsilon)^{s+1} + \nu(\lambda)$. Note that

$$\begin{aligned}
&\Pr[blk_{s+1} = G[k_{s+1}]] \\
&\quad = \Pr[\text{all blocks correct} \mid \text{first } s \text{ blocks correct}] \\
&\quad = \frac{\Pr\left[\text{EXP}^{\text{MPIE}}_{\mathcal{A},\mathcal{O}}(\lambda, m, \epsilon, d, s+1) = 1\right]}{\Pr\left[\text{EXP}^{\text{MPIE}}_{\mathcal{A},\mathcal{O}}(\lambda, m, \epsilon, d, s) = 1\right]} \\
&\quad > \frac{(1-\epsilon)^{s+1} + \nu(\lambda)}{(1-\epsilon)^s + negl(\lambda)}.
\end{aligned}$$

Since s is polynomial in λ, there is a negligible function $negl'$ and a non-negligible function ν' such that

$$\begin{aligned}
(1-\epsilon)^s + negl(\lambda) &= (1 - \epsilon + negl'(\lambda))^s \\
(1-\epsilon)^{s+1} + \nu(\lambda) &= (1 - \epsilon + negl'(\lambda))^{s+1} + \nu'(\lambda)
\end{aligned}$$

Since $(1 - \epsilon + negl'(\lambda))^s < 1$, this means $\Pr[blk_{s+1} = G[k_{s+1}]] > (1 - \epsilon) + negl'(\lambda) + \nu'(\lambda)$. Thus we can construct some $\hat{\mathcal{A}}_1$ that, on input $\{\rho_i\}_{i=1}^m$ runs $\mathcal{A}_1$ and then correctly simulates the environment for the first s queries and finally outputs the files and G'_s. $\hat{\mathcal{A}}_2$ then calls $\mathcal{A}_2$ and discards G'_{s+1}. $\hat{\mathcal{A}}$ thus breaks the single-query incompressibility of the scheme, contradicting our assumption. □

B Proof of Security

We now provide a proof of Theorem 1. We structure this proof as a parallel graph pebbling game. A parallel pebbling game proceeds in rounds: in each round the adversary can place a pebble on each vertex whose parents were all pebbled in the previous round. The adversary then attempts to pebble all outputs of the graph in as few rounds as possible. Each pebble in this game represents data storage, and pebbling a new node represents deriving a new data label. We model $\mathcal{A}$ storing intermediate data as $\mathcal{A}$ placing a bounded number of pebbles in the initial configuration on the graph (i.e. before the first round). The pebbling rules capture data-dependencies in the derivation of new data labels, e.g. enforced by the KDF.

A parallel pebbling game is (s, t, δ)-hard if no adversary starting with s pebbles on the graph can pebble a δ fraction of the output nodes in fewer than t rounds. This intuitively suggests that no adversary can derive more than a δ fraction of the output data labels in t rounds of calls to the KDF starting from an initial state

that stores only s data labels. However, formally proving that there is no adversarial attack that uses less than sm storage (where m is the size of a label) is difficult. It is a long-standing problem to translate pebbling hardness to lower bounds in the random oracle model (ROM), where KDF is a random oracle. Pietrzak [45] recently made progress in this direction, proving that hardness of the parallel pebbling game implies lower bounds in the ROM. However, the analysis is not perfectly tight, leaving open the possibility that the ROM adversary could use a ϵ fraction less storage than the pebbling adversary, where $\epsilon \approx \log n/m$ on a graph with n data outputs.

In our PIE construction based on n-encoder graphs we use *edges* to model data storage instead of vertices. Since the pebbling game places pebbles on vertices, we will need to analyze the pebbling game on a modified graph so that it accurately models the PIE computation. Based on Construction 1, the values along each *data* edge are independent from the values on all other data edges, but the value along each *key* edge is precisely the concatenation of the values of two data edges. This means the values that $\mathcal{A}$ must compute separately are those along data edges. We thus analyze a graph based on the edge-graph on data edges. For each key edge (u, v), we need each data output of u to compute each output of v. We therefore add corresponding key edges in this data-edge-graph. Specifically, if $\mathcal{G} = (V, E)$ with $E = E_D \cup E_K$, we define $\mathrm{DataEdgeGraph}(\mathcal{G}) = (V', E'_D \cup E'_K)$ where

$$\begin{aligned} V' &= E_D \\ E'_D &= \{((u, v), (v, w)) \mid (u, v), (v, w) \in E_D\} \\ E'_K &= \{((u, w), (v, s)) \mid (u, v) \in E_K\}. \end{aligned}$$

Proposition 2. *If $\mathcal{G}$ is a DSaG with ℓ layers and $2n$ inputs, then $\mathcal{G}' = \mathrm{DataEdgeGraph}(\mathcal{G})$'s data edges form a sequence of $\ell - 1$ independent butterfly $2n$-superconcentrators, followed by a single layer of $2n$ vertices. Additionally, key edges are confined to the first layer of each superconcentrator as well as the final output layer, and each such layer forms an independent $(\alpha, \beta/2)$-DRG of key edges.*

Proof. By construction, each superconcentrator in $\mathcal{G}$ is a butterfly n-superconcentrator with two data edges into each input vertex and two out from each output vertex. We see by inspection that the edge graph of this construction is itself a butterfly $2n$-superconcentrator. This demonstrates the vertex and data edge structure claimed.

Now consider the key edges. In the definition of $\mathcal{G}'$, key edges connect to vertices corresponding to the data edges in $\mathcal{G}$ originating from the same vertex as the relevant key edge. As these data edges in $\mathcal{G}$ form the input vertices to a superconcentrator in $\mathcal{G}'$, we see that the key edges are placed as claimed.

To see the depth-robustness, note that for each vertex u in $\mathcal{G}$, there are two data edges originating at u and thus two vertices in $\mathcal{G}'$ corresponding to those edges. Therefore, given any $2\alpha n$ vertices in that layer of $\mathcal{G}'$, those vertices correspond to edges originating from at least αn distinct vertices in $\mathcal{G}$. Additionally, for every key edge (u, v) in $\mathcal{G}$, there is an edge in $\mathcal{G}'$ from each (u, w) to each (v, s). As the corresponding layer of $\mathcal{G}$ was a (α, β)-DRG, there must be a path of key edges in $\mathcal{G}$ of length at least βn containing only those vertices. Thus this must correspond to some path, also of length at least βn in that layer of $\mathcal{G}'$. As they layer has $2n$ vertices, it is thus $(\alpha, \beta/2)$-depth-robust. □

There is one further change to the standard pebbling analysis we must make. Because PRP operations are explicitly invertible, we can place a pebble on a vertex v if either (*i*) every parent of v is pebbled—the regular condition—or (*ii*) there is a pebble on the parent of every key edge and the *child* of ever data edge. This extra condition restricts the types of graphs we can use, but thankfully does not significantly complicate the analysis of a DSaG.

To analyze the security of our PIE construction, we will show that the parallel pebbling game on $\mathcal{G}' = \mathrm{DataEdgeGraph}(\mathcal{G})$, where G is the n-encoder graph in our construction, is $(\delta n, \beta n, \delta)$-hard for every $\delta < 1$. This shows that PIE is tightly incompressible against an adversary restricted to pebbling attacks. Unfortunately, as remarked above, it does not imply d-oracle incompressibility given the current state of pebbling analysis theory, which does not rule out the possibility that a ROM adversary could achieve a small constant compression factor. Nonetheless, there is no known attack, and we will assume this as a conjecture:

Conjecture 1. *If $\mathcal{G}$ is an n-encoder graph and the parallel pebbling game on $\mathcal{G}' = \mathrm{DataEdgeGraph}(\mathcal{G})$ is $(\delta n, d, \delta)$-hard for every $\delta < 1$ then $\mathrm{ENCODE}_{\mathcal{G}}$ (defined by Construction 1) is d-oracle incompressible.*

Theorem 1. *Assuming Conjecture 1, if $\mathcal{G}$ is a (α, β)-DRG with $n = 2^k$ vertices, then the encoding $\mathrm{ENCODE}_{\widehat{\mathcal{G}}}$ defined by Construction 1 on the DSaG $\widehat{\mathcal{G}}$ defined by Construction 2 is βn-oracle incompressible in the random oracle model.*

Proof. We consider the modified parallel pebbling game described above on the graph $\widehat{\mathcal{G}}' = \mathrm{DataEdgeGraph}(\widehat{\mathcal{G}})$. Let D_i be the ith DRG layer—so data edges point toward higher layers—and B_i be the butterfly superconcentrator connecting D_i to D_{i+1}. We assume that for some $\epsilon > 0$, $\mathcal{A}$ stores at most a $1 - \epsilon$ fraction of the data, we begin the game by placing $2n(1 - \epsilon)$ initial pebbles on $\widehat{\mathcal{G}}'$.

Let $U_i \subseteq D_i$ be the set of unpebbled vertices in $v \in D_i$ such that there is an unpebbled path along only data edges from v to some unpebbled vertex in D_ℓ. Note that U_ℓ is thus simply the unpebbled vertices of D_ℓ. Let p_i be the number of pebbles initially placed on $D_i \cup B_i$. Note that $\sum_{i=1}^{\ell} p_i \leq 2n(1 - \epsilon) < 2n$.

Claim 1.1. *$|U_i| \geq 2n - p_i$ for all $i \in [1, \ell]$.*

Proof. We prove this by induction on i, showing that if $|U_{i+1}| \geq 2n - p_i$, then the claim holds for U_i. As our base case, we see that by definition U_ℓ is is the set of unpebbled vertices of D_ℓ, so $|U_\ell| = 2n - p_\ell$.

We now assume $i < \ell$ and $|U_{i+1}| \geq 2n - p_{i+1}$. By assumption, $2n > \sum_{j=1}^{\ell} p_j \geq p_i + p_{i+1}$. Using our inductive hypothesis, this means $|U_{i+1}| \geq 2n - p_{i+1} > p_i$. In particular, this means there is some $U'_{i+1} \subseteq U_{i+1}$ with $|U'_{i+1}| = p_i + 1$.

Let $S = \emptyset$ and let $W \subseteq D_i \setminus S$ such that $|W| = p_i + 1$. B_i is a superconcentrator, so there exist $p_i + 1$ vertex-disjoint paths connecting W to U'_{i+1} using only data edges in B_i. As there are exactly p_i pebbles in this part of the graph and each can be on at most one such path, there is at least one $u \in W$ with an unpebbled path to some vertex in U'_{i+1}. Update $S \leftarrow S \cup \{u\}$. We can now repeat this procedure until $|D_i \setminus S| \leq p_i$, which will first occur when $|S| = 2n - p_i$. Moreover, we note that every vertex in S has an unpebbled path along data edges to some vertex in U_{i+1}, which each have unpebbled paths along data edges to at least one output vertex of $\widehat{\mathcal{G}}'$. Therefore $S \subseteq U_i$, meaning $|U_i| \geq |S| = 2n - p_i$, as desired. □

Claim 1.2. *Assuming $\ell \geq 1/(1 - \alpha)$, there exists an unpebbled path P such that*

1. *P terminates at some unpebbled vertex in D_ℓ*
2. *P has a sub-path of length at least βn containing only key edges.*

Proof. Since $2n > \sum_{i=1}^{\ell} p_i$, by the mean value theorem there is some $i \in [1, \ell]$ such that $p_i < 2n/\ell \leq 2n(1 - \alpha)$. Claim 1.1 shows that

$$|U_i| \geq 2n - p_i > 2n - 2n(1 - \alpha) = 2n\alpha.$$

By Proposition 2, we know that D_i is $(\alpha, \beta/2)$-depth-robust, so therefore there is a key-edge path of length at least $|D_i|\frac{\beta}{2} = 2n\frac{\beta}{2} = \beta n$ touching only vertices in U_i. Since every vertex in U_i contains an unpebbled to

an unpebbled vertex in D_ℓ by definition, the terminal vertex of this key-edge path must connect to D_ℓ, thus proving the claim. □

Claim 1.3. *There exist at least $2n\epsilon$ output vertices of $\widehat{\mathcal{G}}'$ that terminate unpebbled paths P such that:*

1. *Every vertex of P contains an unpebbled path of only data edges to D_ℓ,*
2. *P has a sub-path of length at least βn containing only key edges.*

Proof. We begin with a configuration with $2n(1-\epsilon)$ initial pebbles placed anywhere on $\widehat{\mathcal{G}}'$. Let $O \subseteq D_\ell$ with $|O| = 2n(1-\epsilon)+1$. Now consider a modified configuration that places additional pebbles on all unpebbled vertices of $D_\ell \setminus O$—all *other* vertices in O. Since $|D_\ell \setminus O| = |D_\ell| - |O| = 2n - (2n(1-\epsilon)+1) = 2n\epsilon - 1$, the total number of pebbles in this modified configuration is no greater than $2n(1-\epsilon) + 2n\epsilon - 1 = 2n - 1 < 2n$. Since Construction 2 requires $\ell = \left\lceil \frac{1}{1-\alpha} \right\rceil \geq \frac{1}{1-\alpha}$, Claim 1.2 proves that there is at least one unpebbled vertex in D_ℓ that terminates a long path P with the desired conditions. Because all vertices outside O are pebbled in this configuration, the terminal vertex u of P must be in O.

Since we only *added* pebbles, it follows that u had this property in the original configuration as well. Using the same technique we applied in the proof of Claim 1.1, we can record u in a set S and form the set O' by removing u and replacing it with a different vertex from D_ℓ that has not yet been considered. The same argument applies and we can locate a new $u' \in O'$ with the desired property. We can repeat this argument until $|D_\ell \setminus S| \leq 2n(1-\epsilon)$, which occurs when $|S| = 2n\epsilon$. Each vertex in S has the desired property, proving the claim. □

Every vertex of each path described in Claim 1.3 has an unpebbled path along only data edges to some unpebbled output of $\widehat{\mathcal{G}}'$. Therefore it is impossible to pebble any of those vertices by first pebbling their data-edge children, meaning we must pebble them in topological-sort order, as in a classic pebbling game. Thus in order to pebble each such vertex, it requires at least βn rounds due entirely to key edges. □

C Building a DRG

To construct a DSaG—and thus to implement a PIE using our construction—requires a depth-robust graph. A variety of prior work has explored how to construct DRGs [2, 42] with low α and high β while minimizing in-degree and complexity of graph construction. Despite good asymptotic bounds, none of these graphs provide both proofs of depth-robustness and practical performance.

Mahmoody et al. [42] describe a procedure for building $(\alpha, \alpha - \varepsilon)$-DRGs with in-degree $\widetilde{O}(\log^2 n)$ for any $\alpha \in (0,1)$ and any $\varepsilon > 0$. While such graphs could result in very efficient PIEs in theory, there is not sufficient detail to compute the precise in-degree or implement the construction.

Alwen et al. [2] present two randomized constructions: one has in-degree 2, but $1 - \alpha \approx 1/(4100 \log n)$, while the other has $\alpha > \frac{24}{25}$ and, experimentally, average in-degree $41 \log_2 n - 275$. Even the second of these has $\beta = 3/10$, resulting in an SER of 90. Moreover, both constructions are only depth-robust with high probability and it is not clear how to verify the depth-robustness of a particular graph.

They do, however, provide excellent heuristic security. In particular, the best-known depth reduction attacks indicate that these graphs are very robust in practice. Fisch et al. [32] modify this construction by combining multiple vertices of Alwen's degree-2 graph to produce graphs with any small constant in-degree that appear to be depth-robust in practice. The sampling algorithms are shown in Figure 9. For sampled graphs with n vertices and maximum in-degree 21, they are unable to find an attack that reduces the depth below $n/4$ by removing fewer than $0.58n$ vertices. We therefore heuristically assume that these graphs are

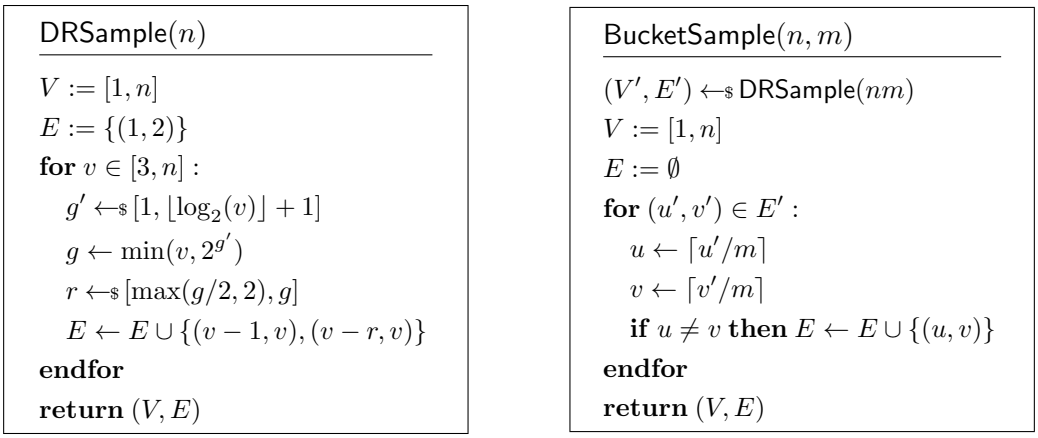


Figure 9: The DRSample algoirthm for sampling degree-2 DRGs due to Alwen et al. [2], and the modification into BucketSample due to Fisch et al. [32]. These algorithms produce very simple low-degree graphs that appear depth-robust in practice.

$(0.5, 0.25)$-depth-robust and use them to provide a practical performance bound in settings where large files are necessary and this security may be considered sufficient.

In order to also measure the performance of a provably-secure PIE, we rely on a naïvely-constructed DRG. Specifically, for any $\alpha \in (0, 1)$, we can build an (α, α)-DRG with n vertices and in-degree $(1 - \alpha)n$. We simply index our vertices from 1 to n, and include edges from vertex i to $i + 1, \ldots, i + \lfloor (1 - \alpha)n \rfloor + 1$. This guarantees that, for any set of k consecutive vertices $i, \ldots, i + k - 1$ with $k < (1 - \alpha)n$, vertex $i - 1$ will connect directly to vertex $i + k$. Thus any set of of vertices $\tilde{V}$ with $|\tilde{V}| \geq \alpha n$ will form a single connected path. This graph's in-degree scales poorly with n, but if we encode large files in multiple independent chunks—as described in Section 6.2—there is no need for large DRGs. Moreover, we see in Section 8 that we can tune KDF to provide very practical timing guarantees on graphs small enough that ENCODE's runtime is dominated by KDF's memory-hardness, not the graph in-degree.

Using the computation from Section 7 to minimize the SER, we set $\alpha = 1/2$.