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
Jacobs Institute, Cornell Tech; IC3†
juels@cornell.edu

†Initiative for CryptoCurrencies & Contracts

Abstract

We present a new primitive supporting file replication in distributed storage networks (DSNs) called a *Public Incompressible Encoding* (PIE). PIEs operate in the challenging public DSN setting where files must be encoded and decoded with public randomness—i.e., without encryption—and retention of redundant data must be publicly verifiable. They prevent undetectable data compression, allowing DSNs to use monetary rewards or penalties in incentivizing economically rational servers to properly replicate data. Their definition also precludes critical, demonstrated attacks involving parallelism via ASICs and other custom hardware.

Our PIE construction is the first to achieve experimentally validated *near-optimal performance*—within a factor of 4 of optimal by one metric. It also allows decoding orders of magnitude faster than encoding, unlike other comparable constructions. We achieve this high security and performance using a graph construction called a *Dagwood Sandwich Graph* (DSaG), built from a novel interleaving of depth-robust graphs and superconcentrators.

PIEs' performance makes them appealing for DSNs, such as the proposed Filecoin system and Ethereum data sharding. Conversely, their near-optimality establishes concerning bounds on the practical financial and energy costs of DSNs allowing arbitrary data.

1 Introduction

The world's data storage demands are ballooning, with annual growth rates of 42% and a projected 50 zettabytes required by 2020 [46]. Supply, however, is lagging [23], 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 [55], Storj [51], MaidSafe, and Filecoin [43] 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, has grown extremely rapidly—over 160 GB in its most efficient standard representation [27]—prompting calls for new decentralized storage architectures [16].

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) [34, 49] and Proofs of Data Possession (PDPs) [8] allow an untrusted server to efficiently prove retention of a file F. Unfortunately, these techniques do not enable a client or verifier to distinguish between an honest provider robustly 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 replication in a restricted *trusted-encoder*, *private-reader* setting where the file owner/client can share secret keys. To store three copies of F, the client simply encrypts F under three different secret keys and uploads the resulting ciphertext triple $G = (C_1, C_2, C_3)$. The provider can then use a PoR/PDP on G to prove 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 [55] and Storj [51] support this approach.

Proving replication is also possible in a *trusted-encoder*, *public-reader* setting. Existing mechanisms [22, 54] show how a file owner can perform a private-key operation using a trapdoor one-way function (RSA) to encode multiple replicas of F. Any entity can recover F from this encoding with a public key, and 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 multiple copies of a file F, yielding a redundant encoding G. (For example, a set of encoders $\{\text{ENCODE}_i\}$ could yield $G = \text{ENCODE}_1(F) \parallel \text{ENCODE}_2(F) \parallel \text{ENCODE}_3(F)$.) 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 randomness is public.*

The public setting is essential for several applications. In Filecoin, for instance, miners prove replicated storage of public files [43]. Similarly, when sharding Ethereum data across multiple providers, no individual owns the data. In both cases providers have an incentive to reduce their storage costs by cheating, and are thus untrusted.

Proving replicated file storage in the public setting is difficult. 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. Moreover, it impossible to *guarantee* that a prover is really storing G directly [29]. For example, the prover might store an encryption of G with key unknown to the client. The ENCODE function can at best allow for detection (and subsequent punishment) of providers who attempt to reduce storage costs.

As we explain in Section 2.1, previous attempts have failed to construct a practical, provably secure encoder in the public setting without relying on implausible assumptions.

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

[1] Other options may be possible. Encodings using very long private keys could incentivize economically rational servers to discard the keys and be unable to recompute encodings on the fly. The feasibility of such approaches is an open research question.

permutations, such as Sloth [36], 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* [7, 25, 30] with the special requirement that it can encode files [29, 41]. 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, providers are monetarily rewarded for periodically proving retention of an encoded file G and delivering it on demand. Such a system could likely create incentives for servers to fully store G as long as they could detect any compression or data loss—precisely what PIEs allow.

We deviate from existing proof of space (PoS) and proof of replication (PoRep) terminology because our concerns, while related, are fundamentally different. First, a PIE is a perfectly-tight PoS *that also allows for data decoding*. Second, PIEs are unnecessary for PoRep systems that allow secret information. Finally, while a public-setting PoRep system would likely use a PIE, it should also prove retention of data over time and, preferably, distribution of data not just replication. PIEs perform neither of these functions.

We therefore believe new terminology is warranted to more precisely describe what our primitive does and does not guarantee.

Our Construction. Our PIE is a graph-based file transformation. It depends on a new construct we call a *Dagwood Sandwich Graph* (DSaG),[2] an iterated interleaving of a *depth-robust graph* (DRG) with a *superconcentrator* [53]. Intuitively, a DRG is a directed acyclic graph that retains a long path even if an adversary removes a many nodes. DRGs are used in memory-hard hash functions [2–4, 13] and to enforce long sequential computations [38], including in proofs of space [25, 30, 41], a purpose we exploit in our DSaG construction. Superconcentrators are graphs with many vertex-disjoint paths between 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 [21]. By sandwiching these two components and iterating, we can prove that any attempt at compression, such as discarding blocks and recomputing them on the fly, forces a storage provider to perform expensively slow and detectable computation.

We show experimentally that, for several realistic and highly tunable concrete timing bounds, our construction is very efficient. With slowness as an explicit goal, we cannot measure performance by simple running time. Instead we develop a new metric called the *Security Efficiency Ratio* (SER), which measures the computational overhead imposed by the encoding beyond the theoretical optimum for a given security bound. By this metric, our construction is only a factor of 4 from optimal. We can additionally improve serial decoding speed by an order of magnitude over typical parameters using Sloth [36]. Our construction operates on small files and scales by achieving high parallelism where it does not damage security.

Limitations. The near-optimality of our construction (under the SER) also supports a negative result. It establishes performance bounds on general-purpose, public-setting DSNs such as Filecoin and reveals fundamental practical limitations. The computational costs needed to secure such systems appear to render them economically infeasible for most data.

Our PIEs, however, show strong practical promise in settings where limitations on the data the DSN can store allow us to avoid these untenable computational costs. These contexts 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. [54].

[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.
- *Implementation and experiments:* We eliminate unnecessary overhead in Section 5 and define the security efficiency ratio (SER), our main efficiency metric, in Section 6. In Section 8 we report experimental performance results.
- *PIE applicability:* In Section 9 we discuss what is needed to build a DSN from a PIE and how the near-optimal performance of our construction provides cost lower bounds that are both concerning for some DSNs and promising for other, more restricted ones.

Section 10 discusses related work and Section 11 concludes.

2 Using Incompressible Encodings

Incompressible encodings alone cannot ensure file storage robustness in a DSN. They do not spread storage across multiple nodes or ensure data availability. They do, however, provide a necessary component for building such a system: *detection* of malicious nodes that use less storage than claimed.

DSNs aim to ensure that a file F is stored with erasure coding or replication so it is 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, thereby providing 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 [8, 34, 49], as noted above, can detect when a server fails to store a target file, but they cannot detect failures of *redundancy*. Suppose three servers S_1, S_2, and S_3 are each supposed to store a copy of F. Even if each proves it can access F, they may *collude* to store only one copy in S_1, and none in S_2 and S_3. When challenged to furnish or prove storage of F, S_2 or S_3 simply access the copy of S_1. The problem, of course, is that if the single copy held by S_1 is lost or damaged, F will be unrecoverable.

Another way to detect deduplication is to *time* responses. 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, is unreliable given the variance in network latency and the ability of servers to co-locate. Public Incompressible Encodings (PIEs), our focus here, provide a slow file transformation that makes response times of cheating servers tunable and independent of the network. PIEs are designed so that a malicious server that has deduplicated data will take much longer to respond than an honest server.

To use a PIE in a DSN, the system transforms a file F into a redundant state, such as replicating (F_1, F_2, F_3) where $F_i = F$, and then uses the PIE to encode this redundant version into a representation G. The DSN servers distribute the pieces of G across multiple servers and store each piece only once, as storing all of G once already contains all necessary redundancy. Consequently, the DSN can verify that F is stored in erasure-resistant fashion by simply verifying that each piece of G is present. Other applications of PIEs, such as proving files bear watermarks [54], look similar.

Note that PIEs are only one component of a DSN. A PIE itself only ensures that correctly-encoded data cannot be compressed. A full system must ensure that data is encoded correctly, distribute data to different servers, and incentivize servers to store it. Systems with financial rewards can levy penalties for *not* storing

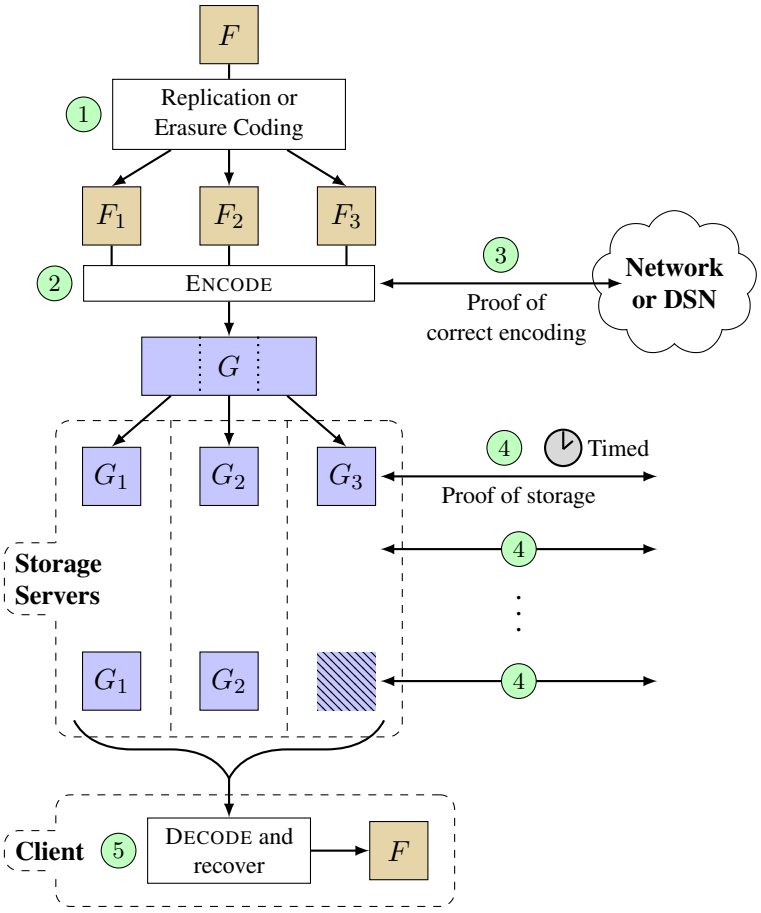


Figure 1: The basic structure of a DSN using a PIE to ensure data robustness. (1) First, file F is replicated or erasure-coded to ensure robustness. (2) The server runs this robust data through ENCODE yielding incompressible representation G. (3) The network/DSN queries the server to prove the data was encoded properly. (4) The network/DSN repeatedly queries the servers storing the pieces of G 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 is lost or corrupted the DSN can detect this and the client can still recover F.

data (see, e.g., Filecoin [43]), but how to set and levy those fines and ensuring that data is distributed across multiple servers [11, 15] is beyond the scope of this work.

We also provide no new efficient means to verify that G is a correct encoding of F. In some sense this is trivial. Anyone can encode or decode the data, so anyone can check that $G = \text{ENCODE}(F)$. Requiring every node to validate every encoding, however, would incur prohibitive overhead. We discuss some directions in Section 9.1 that incorporate prior work and depend on the DSN structure.

2.1 Previous Approaches to PIEs

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

Proofs of Space. PIEs are closely related to *Proofs of Space* (PoS) [25], interactive protocols in which a prover publicly commits to using some amount of space and then responds to challenges. Soundness 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 include verifi-

cation of the encoding correctness, while we separate this problem from proving the incompressibility of a correct encoding. 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, 41], in which an adversary saving any ϵ fraction of space must fail a random challenge with probability proportional to ϵ (for PIEs, that proportion is 1). 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 perfectly tight PoS can be used to construct a PIE. A generic construction of Pietrzak [41], 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 [37] suggests using "time-asymmetric encodings," such as a Pohlig-Hellman cipher [42] based on modular exponentiation, to apply a slow transformation to each input file block using a unique identifier as a key. Boneh et al. [12] generalize this construction as *decodable verifiable delay functions*. Intuitively, this yields a sound PIE—re-deriving any block of the encoding is slow—but 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 6.

The Filecoin [43] project introduced the term *proof of replication* to codify the desired properties of a primitive for their DSN. They propose using this primitive to both encode data and provide Sybil resistance for a blockchain by replacing proof-of-work with proof-of-space. Their initial technical report [10] proposed interleaving multiple chained (CBC) encryption passes under a public key κ with block permutations. Unfortunately, previous work [54] shows an attack in which a cheating prover checkpoints specific intermediate blocks of the encryption as "shortcuts" to quickly recompute discarded output. It 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 [54] 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, they assume slow retrieval of F due to use of rotational hard drives, which is clearly invalid given the proliferation of fast solid-state drives (SSDs) and monetary incentives to cheat in systems like Filecoin.

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 randomness 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 ρ is 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 the *incompressibility* of a PIE with a game between the encoder and a public challenger. For any adversary $\mathcal{A}$ and set of (not necessarily distinct) files $\{F_i\}$, $\mathcal{A}$ is given $\{F_i\}$ and public randomness $\{\rho_i\}$ and allowed to perform any preprocessing it would like. We then encode to some $G = \{G_i\}$ using $\{\rho_i\}$ and challenge $\mathcal{A}$ 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 $F_1 = F_2 = F$, a cheating $\mathcal{A}$ could just store F and compute blocks of $G = (G_1, G_2)$ on demand when challenged.

In practice we can avoid this impossibility by making ENCODE 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 application-specific integrated circuits (ASICs) [50], which achieve modest improvements in sequential performance and drastically increased parallelism. This security notion has been used before in time-lock puzzles [45], proofs of sequential work (PoSW) [18, 38], verifiable delay functions (VDFs) [12], 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. Indeed, this technique has been proven to require sequential work in the parallel random oracle model [38, 41].

We follow this approach and limit the number of times an attacker can execute an intentionally slowed key derivation function modeled as a random oracle $\mathcal{O}$. We constrain the number of sequential calls to $\mathcal{O}$ by defining a parallel oracle $\mathcal{O}^\star_d$ 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 $\mathcal{O}^\star_d$ as follows, initializing $c = 1$:

$\mathcal{O}^\star_d(x_1, x_2, \dots)$

if $c \geq d$ **then abort**
$c \leftarrow c + 1$
return $(\mathcal{O}(x_1), \mathcal{O}(x_2), \dots)$

We now define security in terms of an adversary who must return encoded file blocks given limited storage and access to $\mathcal{O}^\star_d$. 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 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 $\epsilon \leq 1$ 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

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

1 : $\{\rho_i\}_{i=1}^m \leftarrow\!\!\$\ \{0,1\}^{\lambda \times m}$
2 : $G' \leftarrow\!\!\$\ \mathcal{A}_1^{\mathcal{O}}(\{(F_i, \rho_i)\}_{i=1}^m)$
3 : **for** $i \in [1, m]$: $(G_i, \kappa_i) \leftarrow \text{ENCODE}^{\mathcal{O}}(F_i\,;\,\rho_i)$
4 : $G := G_1 \,\|\, \cdots \,\|\, G_m$
5 : $j \leftarrow\!\!\$\ [1, |G|]$
6 : $blk \leftarrow\!\!\$\ \mathcal{A}_2^{\mathcal{O}^\star_d}(j, G', \{(F_i, \rho_i, \kappa_i)\}_{i=1}^m)$
7 : **return** $\left(\frac{|G'|}{|G|} \leq 1 - \epsilon\right) \wedge (blk = G[j])\,.$

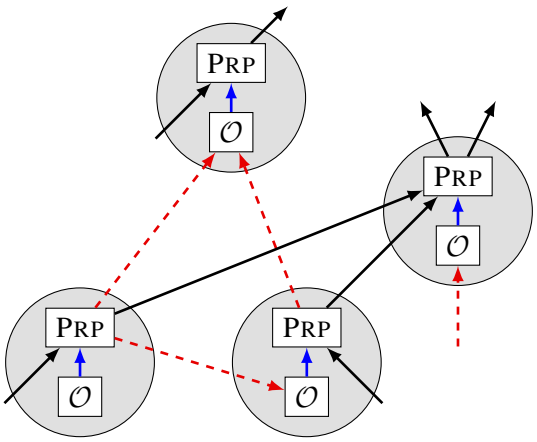


Figure 2: The high level structure of an encoding based on a DAG. Solid black edges are data edges, dashed red edges are key edges, and blue arrows from $\mathcal{O}$ to PRP show the use of derived keys.

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 [54] 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. We begin with an informal overview.

To satisfy our security definition and catch a cheating attacker, our encoding will require a large number of inherently sequential calls to some moderately slow operation. As we will see later this can either be a key derivation function KDF or a pseudorandom permutation PRP. For now we will use a KDF, which we model as a random oracle $\mathcal{O}$.

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 $\kappa_i = \mathcal{O}^{(d)}(i \parallel F[i])$ 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. Since both are the same size, both require at least $|G|$ storage, making the encoding incompressible. This approach, however, is extremely expensive, as it requires nd oracle calls to achieve d-oracle incompressibility for an n-block file.

To reduce the cost without damaging security, we aim to amortize the inherently slow work across multiple file blocks. Optimally, recovering a single discarded block would require the same amount of computation as encoding the entire file. We describe below how to get within a small constant factor of this ideal.

Intuitively, this amortization requires entangling the encodings of different blocks in the file, making them rely on each other. We do this by following prior and concurrent work on proofs of inherently sequential work [19, 38] and proofs of space [7, 25, 30, 41, 44] and use a directed acyclic graph (DAG) to define our encoding. Each vertex in the graph defines a computation that is dependent on each of that vertex's parents.

Proofs of space typically derive *labels* for each vertex by hashing the labels of its parents. By analyzing the graph, we can determine the level of interdependency between the encodings of different blocks of the file. Specifically, paths in the DAG result in inherently sequential work during encoding.

Using only non-invertible hash functions, it is unclear how to support rapid decoding, which we consider critical, so we add extra structure to our graphs. This 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; we must be able to recover blocks passed along data edges. We therefore use a keyed psuedorandom permutation PRP_κ for these transformations. Key edges represent (lossy) dependencies used to determine keys, and thus how other data is transformed. Figure 2 depicts how we call $\mathcal{O}$ on all incoming data from key edges and use the resulting key κ to permute data passed in along data edges with PRP_κ.

We now discuss what this means formally and what properties a DAG needs to define a valid encoding. We then detail our specific graph construction and how it guarantees incompressibility.

4.1 Encoder Graphs

The DAG-based data transformation described above is not always invertible. Undoing the operation at a given vertex requires the values assigned to *outgoing* data edges and *incoming* key edges. We can define a decoder graph representing these dependencies by reversing the direction of appropriate edges. If this decoder is a DAG, the transformation is invertible. A cycle in the decoder graph, however, 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 graph)**.** 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

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

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.

Condition 3 formalizes the intuition about when decoding is possible. We provide detail about how to execute decoding after formalizing encoding.

While Definition 2 is far from totally general—it does not allow encoding to condition on data, for example—it provides a sufficient condition that we meet.

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_i, we invoke $\mathcal{O}$ on all incoming key edges to derive a key κ_i. We then permute the concatenation

of all incoming data edges using a pseudorandom permutation (PRP) keyed by κ_i. The result is sent out along all outgoing key edges and partitioned among outgoing data edges.

Formally, we let $\textsc{Kdf} : \{0,1\}^* \to \{0,1\}^\lambda$ be a key derivation function modeled as a random oracle, and $\textsc{Prp}_\kappa : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a keyed PRP modeled as an ideal cipher. We assume for simplicity that a evenly divides λ. Using this notation, Construction 1 specifies how to compute $\textsc{Encode}_{\mathcal{G}}(F\,;\,\rho)$.

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 $F = F[1] \,\|\, \cdots \,\|\, F[n]$ where each $F[i]$ is $\frac{\lambda}{a}$ bits, 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 $y_1, \ldots, y_a$ be the values assigned to incoming data edges.

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

  Output y' along each outgoing key edge. For the a outgoing data edges $e^i_1, \ldots, e^i_a$ ordered by the topological sort of their terminal vertices, output y'_j along e^i_j.

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

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

Decoding. Decoding a file requires reversing the computation described in Construction 1. For an arbitrary graph this may be infeasible because computing κ_i may require data we cannot access. For encoder graphs, however, Condition 3 of Definition 2 ensures that all information is available.

For vertex v_i in $\mathcal{G}$, we must compute $y_1 \,\|\, \cdots \,\|\, y_a = \textsc{Prp}^{-1}_{\kappa_i}(y')$. This requires both y' and the ability to compute κ_i. To compute y', we need the values originally output along all data edges emitting from v_i. That is, we must decode each vertex u with $(v_i, u) \in E_D$. To compute κ_i, we need the values of all incoming key edges. That is, for each $(w, v_i) \in E_K$, we need the value output when computing w *during encoding*. During decoding, those values are output by decoding the data-edge children of w. Therefore, if $(w, w') \in E_D$ and $(w, v_i) \in E_K$, we must decode w' before we can decode v_i.

The edges of $\mathcal{G}^{-1}$ defined in Condition 3 of Definition 2 are precisely the computational dependencies described above. If $\mathcal{G}^{-1}$ is acyclic, ordering the vertices based on a topological sort of $\mathcal{G}^{-1}$ and decoding them in that order ensures that we have already computed y' and $x_1, \ldots, x_k$ as defined by Construction 1 before we try to decode v_i. We can then simply compute $\kappa_i = \textsc{Kdf}(\kappa \,\|\, i \,\|\, x_1 \,\|\, \cdots \,\|\, x_k)$ and $y = \textsc{Prp}^{-1}_{\kappa_i}(y')$, exactly inverting the encoding operation. While there is no requirement that this decoding order relate to the encoding order in any particular way, for our construction, and others that are straightforwardly invertible, it is exactly the reverse of the encoding order.

Lastly, because we simply reversed the direction of all data edges in $\mathcal{G}$, the source vertices of $\mathcal{G}^{-1}$ are precisely the sink vertices of $\mathcal{G}$.[3] Since these vertices are the outputs of ENCODE, the values required to initialize the above algorithm—the inputs to source vertices of $\mathcal{G}^{-1}$ and κ—are exactly the inputs provided to DECODE.

Note that if $\mathcal{G}$ is not data a-regular, the above technique works with a minor modification: it requires families of KDFs and PRPs to operate on each data size that appears at a vertex.

[3] Vertices in $\mathcal{G}$ with no connected data edges neither produce nor consume data and can therefore be ignored.

4.3 Guaranteeing Sequential Work

For any encoder graph, Construction 1 always produces an invertible encoding, but it does not always produce a PIE. A PIE requires that an adversary $\mathcal{A}$ must make many inherently sequential calls to $\mathcal{O}$ to recompute discarded data. As the edges in the encoder graph $\mathcal{G}$ represent data dependencies, paths in $\mathcal{G}$ translate to inherently sequential operations in $\text{ENCODE}_{\mathcal{G}}$. 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 prohibits $\mathcal{A}$ from recomputing discarded data efficiently even if it stores *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.

If $\mathcal{A}$ is storing intermediate data, that data must correspond to one or more vertices in $\mathcal{G}$. We can model this by marking, or *pebbling* those vertices. An unpebbled path ending in a sink node then represents an inherently sequential computation that $\mathcal{A}$ must perform to recompute a discarded data block. We employ this technique formally in Appendix B to prove our main security result.

Depth-robust graphs. To guarantee long paths in similar adversarial settings, much prior and concurrent work [25, 30, 41] 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 to construct proofs of space, memory-hard functions [2–4, 13], and proofs of inherently sequential work [38]—a use very similar to ours and 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 and sink nodes for each existing vertex connected via data edges. The resulting encoding, however, is not a PIE. $\mathcal{G}$ is a DAG, so an adversary $\mathcal{A}$ can simply discard the output produced by a source node of $\mathcal{G}$ and recompute it very rapidly. Pietrzak [41] showed how to overcome this attack to construct a tight proof of space by only considering the outputs from a subset of the vertices. Unfortunately, when applied as a PIE, this technique would require an honest decoder to recompute any part of $\mathcal{G}$ not used as output, 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. While intuitively sensible—if $\mathcal{A}$ discards data there should always be an unpebbled long path somewhere—this may not be sufficient. If $\mathcal{A}$ discards an ϵ fraction of blocks, we can guarantee that an unpebbled long path exists in some copy of $\mathcal{G}$, but the end of that path may not 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, the end of any unpebbled path of key edges connects to some vertex on the next layer, which connects to a vertex on 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 [10]—appears to create the required connection, but any reasonable number of passes is succeptible to "shortcut" attacks, as discussed in Section 2.1.

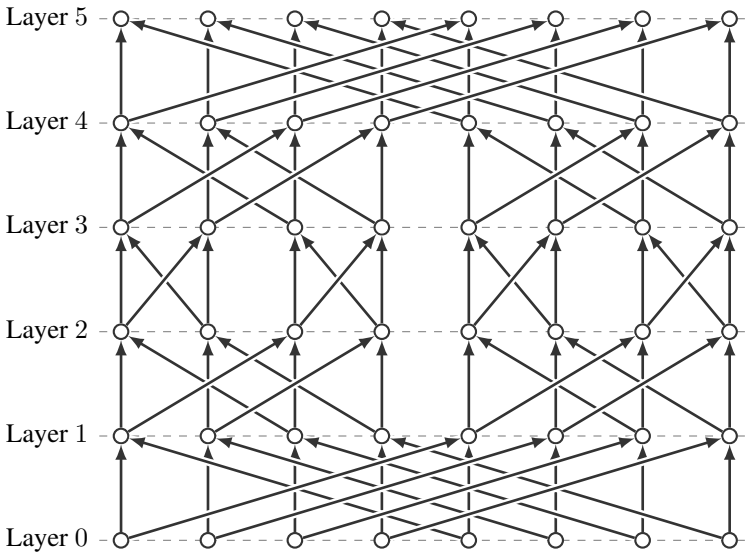


Figure 3: $\mathcal{B}_3$, a butterfly superconcentrator on 8 elements.

4.4 Dagwood Sandwich Graphs

There is a much more robust connector than a series of CBC passes: a *superconcentrator* [53]. Superconcentrators are graphs where any subset of r inputs and r outputs are connected by r vertex-disjoint paths. This means severing connections between inputs and outputs requires just as much data as storing the corresponding outputs. Their formal definition is as follows.

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'.

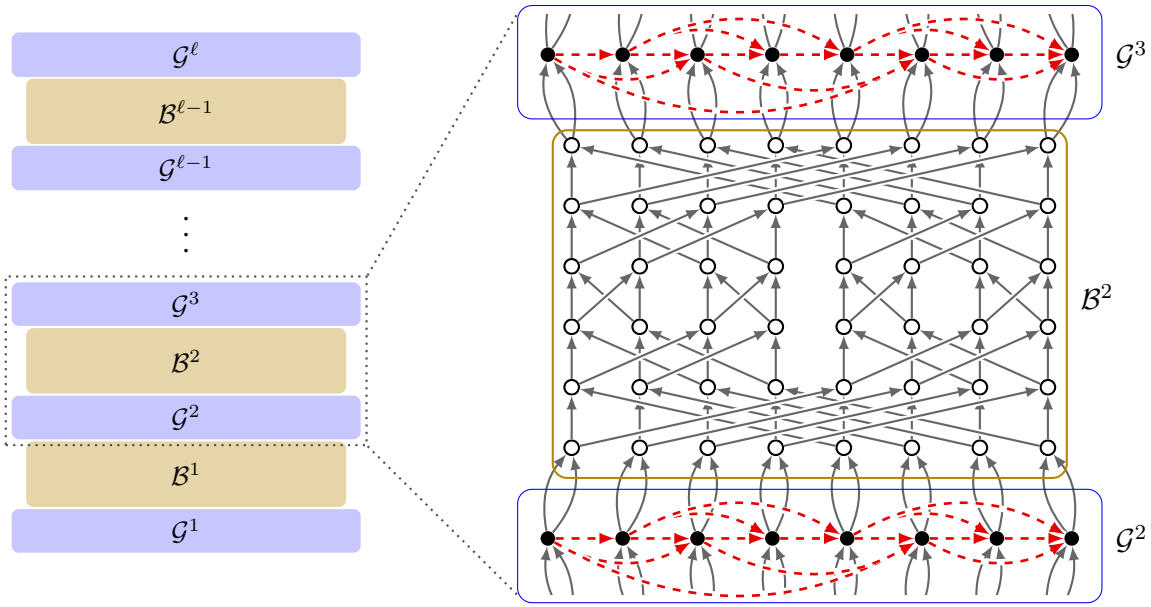
Perhaps the simplest construction is the *butterfly superconcentrator*, created by connecting two classic butterfly graphs. These graphs have the added benefit of being 2-regular—all (non-source/sink) vertices have both in and out-degree 2. The number of inputs and outputs of such a graph must be a power of 2, so we let $\mathcal{B}_k$ be the butterfly 2^k-superconcentrator. Figure 3 shows $\mathcal{B}_3$.

This construction appears as a good candidate to connect our layers of DRGs, and 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.

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 $\mathcal{B}_k$. For each vertex in $\mathcal{G}^1$, connect two new vertices to it that are inputs of $\widehat{\mathcal{G}}$. For each vertex in $\mathcal{G}^\ell$, connect it 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.



(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 ($\mathcal{B}^i$). The right shows the edge structure, with data edges in solid gray and key edges in dashed red.

With $\left\lceil \frac{1}{1-\alpha} \right\rceil$ layers, no adversary $\mathcal{A}$ can eliminate all long paths in all copies of $\mathcal{G}$ using less data than the full file size. The vertex-disjointness condition in Definition 4 further prevents $\mathcal{A}$ from gaining any advantage by storing data inside the connectors. Construction 2 therefore defines a data 2-regular $2n$-encoder graph that defines a PIE using Construction 1 when the adversary is restricted to attacks that follow a pebbling game. The current state of pebbling analysis theory unfortunately does not rule out a ROM adversary achieving a small compression advantage. Nonetheless, there are no known attacks, so we assume as a conjecture that none exist.

Conjecture 1 (Informal)**.** *If $\mathcal{G}$ is an n-encoder graph and no adversary beginning with k or fewer pebbles on $\mathcal{G}$ can pebble more than k output vertices in fewer than d rounds, then* $\textsc{Encode}_{\mathcal{G}}$ *(defined by Construction 1) is d-oracle incompressible.*

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

We provide a more formal version of Conjecture 1 and a proof of Theorem 1 in Appendix B.

5 Improving Efficiency

The DSaG construction provides strong security, but in its simplest form is highly inefficient. There are, however, several ways to significantly improve performance.

5.1 Rapid Decoding

One critical performance component is the *decoding* speed, and our construction provides two routes to make it fast. 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, allowing 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. As a result, outputs of encoding some vertices are needed to compute the keys used to encode (and decode) other vertices *in the same layer*. The values output by a vertex during encoding are, however, the values *input* to that vertex during decoding. This means that, in order to 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 compute the data transformation for every vertex in a given DRG layer simultaneously, massively reducing the required wall-clock time.

Faster Sequential Decoding. Using an *asymmetric-speed PRP*, we can significantly hasten even sequential decoding. 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 but 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 [36] based on modular square roots interleaved with a fast PRPs. Boneh et al. [12] note that separately applying such a PRP to each small 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, with only minor overhead in overall work.

As we show in Section 8.3, both of these rapid decoding 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 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 $\text{ENCODE}_{\mathcal{G}}$ 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.

5.3 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 needed for a PIE. This is because 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 do not need to generate different graphs for different files; one global graph is sufficient. The size of that graph determines the chunk size and thus 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 slow operation 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 $O\left(|\mathcal{G}||F|/n\right)$—linear in $|F|$.

6 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.

SER is an empirical measure, so it accounts for computational overhead without security benefit and varies with parallelism for a PIE that allows it during encoding. While it is possible to cut a file into chunks and parallelize encoding across chunks (Section 5.3), DSaGs preclude parallelism within a file chunk, so we only measure the SER for a single chunk.

We can also lower bound the SER based on the DRG parameters to help choose which graph to use. Ideally the cost of ENCODE will be dominated by the number of calls to KDF. Using the fast KDF′ optimization from Section 5.2, 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 α and β are constants—a common goal of DRG constructions—this lower bound 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 $\alpha = \frac{1}{2}$, providing a lower bound of 4 for the SER.

The actual SER may be higher 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. As we see in Section 8, we are able to achieve SERs extremely close to this bound for small files and when using a heuristically-secure DRG.

7 Building a DRG

Constructing a DSaG—and thus to implement a PIE using our construction—requires a depth-robust graph. A variety of prior work has explored building DRGs [2, 38] 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. [38] provide a theoretically-efficient construction for $(\alpha, \alpha - \varepsilon)$-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 related randomized constructions: one has in-degree 2, but $1 - \alpha \approx 1/(4100 \log n)$, while the other has $\alpha < \frac{1}{25}$ and, experimentally, average in-degree $41 \log_2 n - 275$. Even the second of these has $\beta = 3/10$, resulting in an SER over 80. 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.

To measure the performance of a provably-secure PIE, we therefore rely on a naïvely-constructed DRG. Specifically, for any $\alpha \in (0, 1)$, we can easily build an (α, α)-DRG with n vertices and in-degree $(1 - \alpha)n$ (see Appendix C.1). Unfortunately, as we would expect, this linear in-degree results in poor scaling on large files. We can encode large files in multiple independent chunks to alleviate the need for large DRGs (see Section 5.3), but more efficient DRGs are still desirable.

Despite the poor proven bounds, the Alwen et al. construction appears to effectively resist known depth-reduction attacks in practice, providing evidence of highly scalable efficiently computable DRGs. A slight modification by Fisch et al. [32] produces graphs with small constant in-degree that effectively resist known depth-reduction attacks. In particular, they found that a maximum in-degree of 21 was sufficient to prevent known attacks from reducing the depth of an n-degree graph below $n/4$ when removing fewer than $0.58n$ vertices. Given these results, we additionally provide benchmark results for these graphs, heuristically assuming they are $(0.5, 0.25)$-depth robust. We provide the exact construction algorithm for convenience in Appendix C.2.

8 Experiments

We now present performance results for our PIE implementation. Our PIE implementation is 650 lines of Java code that rely on BouncyCastle [14] for all hash and cipher primitives. For all benchmarks, we let $\lambda = 512$ be the block size. We make this choice because smaller blocks result in more operations, slowing performance, while most common and efficient hash functions and block ciphers operate on at most 512-bit values. Except where otherwise noted, we ran all benchmarks on a `c5.large` Amazon EC2 instance, which provides 2 cores of an Intel Xeon Platinum 8124M CPU.

We benchmark the two constructions described above. For the naïve DRG, we set $\alpha = \frac{1}{2}$ to minimize SER based on the computation in Section 6. While we get a near-optimal SER of 4 on small file sizes, the linear in-degree results in poor performance on large file chunks. We can break large files into smaller chunks to avoid this overhead (see Section 5.3), but doing so limits our ability to amortize the slow operations over larger amounts of data. Here we use only fast PRPs and instantiate a slow KDF as scrypt [40]. We choose scrypt due to its popularity and tunable performance. Its memory-hardness may appeal to some applications, but any tunable-speed cryptographic hash function would produce the same results in our benchmarks.

For the heuristically-robust graph specified by Fisch et al. [32], we assume it is $(0.5, 0.25)$-depth-robust based on Fisch's experimental results. This provides an SER bound of 8 and much better scaling, allowing us to encode more data with the same total amount of computation for a given security bound. To enable extra-fast decoding, we use only fast KDFs and employ Sloth [36] as an asymmetric-speed PRP. We provide more detail on these constructions in Appendix 7.

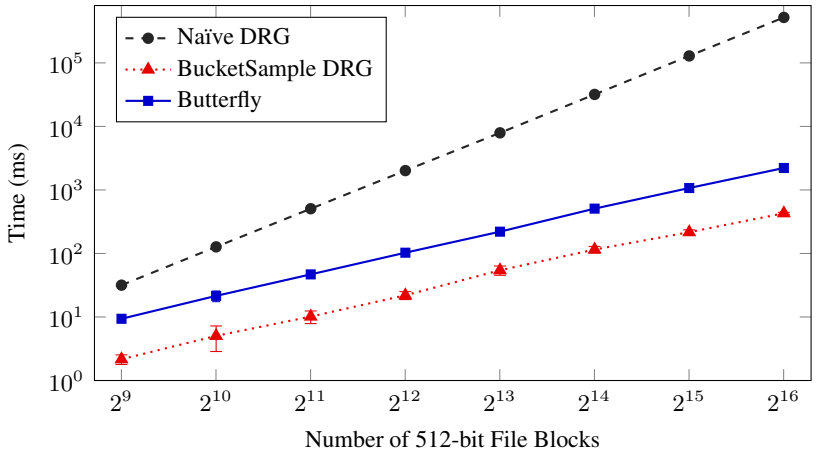


Figure 5: Fast operation overhead for different layer types and sizes. Naïve DRG run time grows quadratically with the graph size, while the others grow roughly linearly.

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 as discussed in Section 5.2.

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 the butterfly superconcentrator and the DRG transformations within our PIE. For each transform, we measured the run time for a variety of graph sizes. SHA512's compression algorithm is far faster than scrypt's, so even when using scrypt in the DRG layer we first compress the key inputs using SHA512. We therefore include SHA512's compression time when measuring this overhead, 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 performance scales poorly due to its linear degree.

8.2 ENCODE Performance

To ensure that the formal security of a PIE corresponds to practical security, our "slow" operation to be sufficiently slow. Specifically, 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 5.3. 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 shows 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.

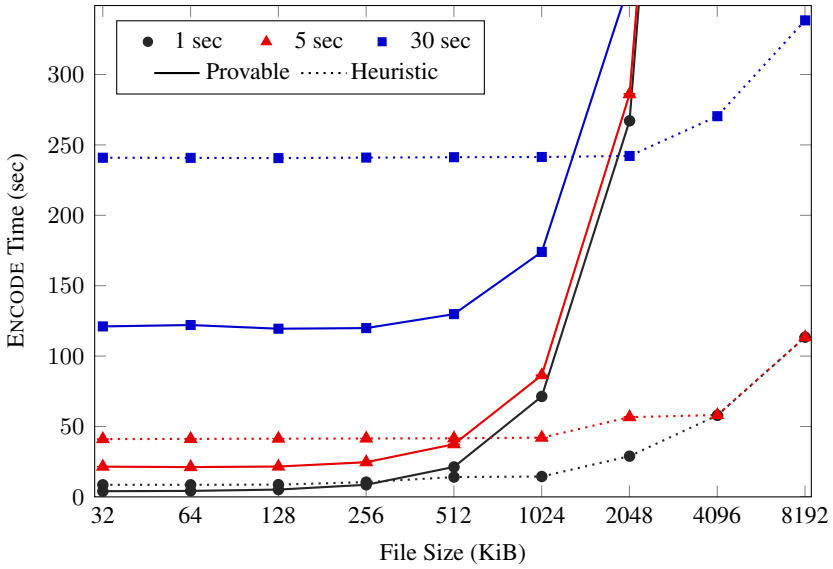


Figure 6: ENCODE time by file size for wall-clock security bounds of at least 1, 5, and 30 sec. Solid lines use scrypt as a slow KDF and a provably-secure naïve DRG that scales poorly to large files. Dotted lines use a heuristically depth-robust graph and Sloth as a slow PRP. These scale to large files, but Sloth's fastest configuration takes 420 μs per 512-bit operation, limiting tuning. For example, security bound for a 4 MiB file must be a multiple of 6.8 second. Overhead of "fast" operations remains minimal.

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.large`. 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.

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. When using Sloth, single-threaded decoding is fast enough that parallelism provides little benefit. With a 5-second security bound, increasing the thread count provides no noticeable benefit beyond 5 threads.

Finally, we benchmarked single-threaded DECODE performance with Sloth for a variety of file sizes. Figure 8 shows that 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. For larger files, however, we tune Sloth to be faster on a single invocation, making overhead from larger graphs noticeable, causing decoding times to scale linearly with graph size.

We modify Sloth's difficulty by iterating its basic asymmetric-speed permutation. The performance asymmetry on 512-bit blocks is over a factor of 200, resulting in the same gap between ENCODE and

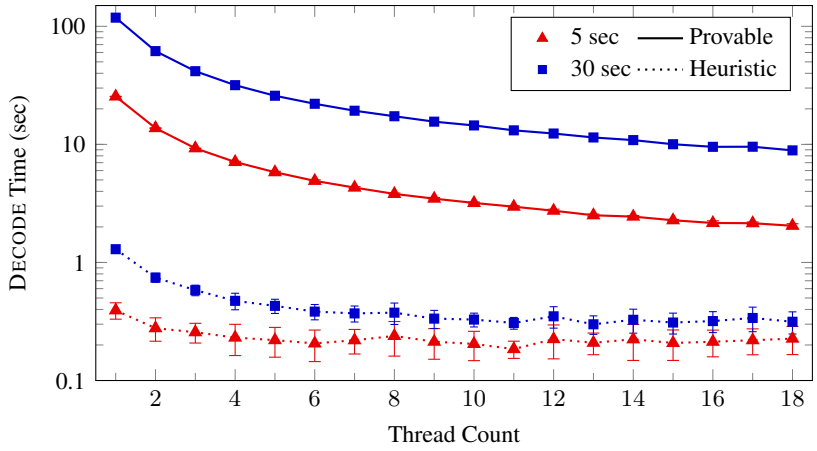


Figure 7: DECODE time for a 256 KiB file for both 5 and 30-sec security bounds run with multiple threads on an Amazon EC2 `c5.18xlarge`. Solid lines use a provably-secure naïve DRG and scrypt. Dotted lines use a heuristic DRG and Sloth.

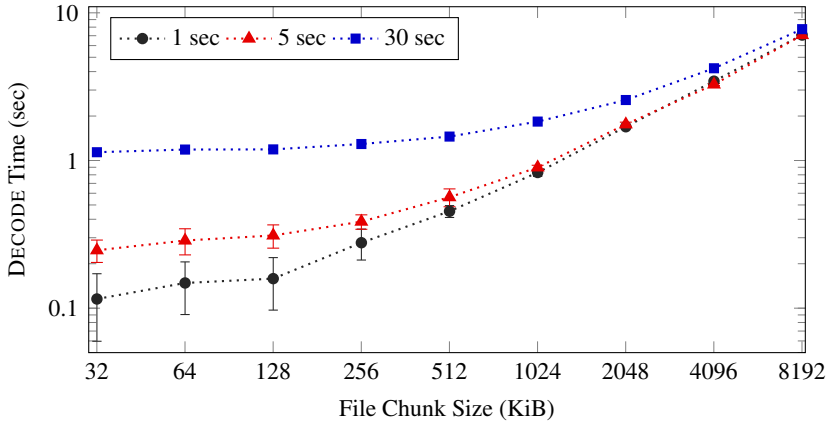


Figure 8: DECODE time by file size for 1, 5, and 30-sec security bounds using a heuristically-secure DRG and the Sloth PRP.

DECODE when Sloth dominates both run times. For example, on a 32 KiB file, Sloth with 576 iterations required 241 sec for ENCODE, but only 1.1 sec for DECODE. With fewer iterations, the overhead of fast symmetric-speed operations is relevant. With one iteration per block—Sloth's fastest tuning—DECODE is roughly 16× faster than ENCODE. Parallelism, though less beneficial than with scrypt, is still effective. With Sloth set to 72 iterations, we decoded a 256 KiB file using 13 threads in 300 ms—800× faster than ENCODE.

9 Practical Lessons and Application

Rapid decoding appears critical for Content Distribution Network (CDN) style DSNs like those proposed by Filecoin and needed to shard Ethereum data. Lengthy and computationally-intensive decoding would limit use to little more that data backups. Section 8 indicates promising performance for Ethereum-like systems that can restrict the data being stored. Unfortunately, the rapid decoding itself raises fundamental questions about the cost more general systems like Filecoin, especially when compared to commercial storage providers like Amazon S3.

Careful readers will note that Definition 1 for incompressibility is only secure under composition—

parallel or sequential—if the files in one execution of $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}$ are independent of other executions. Without this guarantee composition completely breaks security.

Consider composing two runs of one file each. In the first run, F_1 is an arbitrary file that correctly encodes to G_1 with randomness ρ_1. In the second, we let $F_2 = G_1$ and $G_2 = \text{ENCODE}(G_1\,;\,\rho_2)$. 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 [41], it is not a property of our PIE scheme in particular.*

There are two clear ways to address this concern. The first is to only accept data trusted to be generated independently of existing encodings. For example, blockchain state data is honest by assumption. Such a setting can reap the benefits of both nearly optimal encoding times and highly efficient decoding, which we believe would make our PIEs a highly practical primitive.

Ethereum, for example, averages roughly 225 MB of new state per day [27]. As the fastest configuration of our Sloth-based PIE requires at least 14.5 sec/MB of CPU time on a `c5.large` Amazon EC2 instance,[4] the single-time encoding cost for each replica would be roughly 1.7¢ per day using Amazon EC2 spot instances.[5] Reducing the replication by as little as a factor of 10 would thus be cost-efficient in one year unless ongoing storage costs were below 0.064¢ per GB monthly—less than Amazon S3's cheapest offering.[6]

The second way to avoid composition attacks requires storage providers to commit to their files before selecting randomness, reducing security to a single execution of $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}$. This imposes little overhead when files are static and encoded simultaneously (e.g., infrequent backups), but because additions or modifications may be conditioned on existing encodings, we must re-encode *all* files for *any* change. With no other general mechanism for showing generic files are independent, this appears to be an inherent limitation.

Limitations for DSNs. In settings such as Filecoin which both allow new data to enter the system and account for adversarial files, the DSN must operate in epochs. Following the second approach above, data remains static within an epoch, and providers commit to and re-encode all data at the beginning of a new epoch. As discussed in Section 6, 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. Using the same parameters as above, an honest provider must pay 7.8¢/GB for each encoding. Note that this is the cost for one replica; triplicate replication would impose 3× overhead. Using epochs of one day, for example, would balloon the monthly cost to $2.34/GB for each replica, simply for re-encoding. Compared to commercial services like Amazon S3—priced at 2.3¢/GB normally and less than 0.1¢/GB for "Glacier Deep Archive" storage [6]—these costs are enormous. Moreover, this only accounts for computational overhead; it does not consider data access costs, which may be substantial.

These prices not only raise serious concerns about the cost-effectiveness of such a scheme, they also indicate a very high energy consumption. One claimed motivation for systems like Filecoin is to provide a Sybil-resistant consensus mechanism that is less wasteful than Bitcoin-style Proof of Work. The need for frequent computationally-intensive encodings threatens to eliminate much of that benefit, as the Sybil-resistance provided by the PoS aspect of a PIE would require immense quantities of computation.

9.1 Ensuring Correct PIE Computation

Our main results focus entirely on ensuring that a correctly-encoded PIE cannot be compressed. As we mentioned in Section 2, the public setting requires validation of the encoding itself. While anyone can check that an encoding is correct—potentially efficiently if decoding is sufficiently fast—providing a compact proof is nontrivial.

[4] As this uses the fastest Sloth setting, larger file chunks would increase the minimum sequential work, but not reduce the total time per byte.

[5] Amazon EC2 `c5.large` spot instances currently cost 1.9¢ per hour [5].

[6] "Glacier Deep Archive," which currently costs 0.099¢ per GB monthly [6].

There are multiple ways to address this concern depending on the setting. These differences are why we choose to separate the proof of correct encoding from the guarantees provided by a correctly-encoded PIE. For example, in a permissioned or hybrid consensus system [39] with small committees presumed to contain a quorum of honest nodes, such a quorum could verify encodings. In permissionless settings, we can rely on the existing state of the art for proofs of space [25, 30, 44]: graph sampling.

Graph sampling is only able to guarantee with high probability that a fixed percentage of the vertices were computed correctly with respect to their parents. Unfortunately, a small number of incorrect vertices may escape detection, allowing correspondingly small compression. How to close this gap and provide overall systems with tighter proofs of space remains an open question.

One potential approach utilizes the fact that the public setting allows anyone to generate a compact proof, via Succinct Non-interactive ARguments of Knowledge (SNARKs) or refereed delegation of computation [9, 17, 52], that an encoding is, with overwhelming probability, *incorrect*. If the network financially rewards whistleblowing and punishes dishonest storage servers, it may be able to remove incentives for storage servers to compute *any* part of an encoding incorrectly. Analysis of such incentives would likely be nontrivial, but such a mechanism would apply to other PoS schemes as well.

10 Related Work

Proofs of Storage. As noted above, PoRs [24, 34, 49] and PDPs [8], also called proofs of storage [35], 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, as may multi-replica PDPs, e.g., [20].

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

Proofs of Replication. Several early-stage DSNs, such as Sia [55] and Storj [51], already perform proofs of replication in the trusted-encoder, private-reader setting.

Van Dijk et al. [54] propose an encoding technique using RSA trapdoor one-way permutations not intended for, but useable for, proving replication in the trusted-encoder, public-reader setting. Damgård, Ganesh, and Orlandi [22] propose a similar construction, but with PoRs to support file extraction, rather than just incompressibility, and an analysis focused on file replication.

Beyond the previous attempts to construct PIEs [10, 43, 54] noted above, Fisch et al. [31] first proposed using DRGs to address the untrusted-encoder, public-reader file replication problem. In independent concurrent work, Fisch proposed a security notion for proof-of-replication with rational adversaries called ϵ-rational replication, showing PIEs both necessary and sufficient [29]. He also constructs an alternative PIE from bipartite expander graphs instead of superconcentrators [30].

Unlike our construction, which provides optimal compression resistance and near-optimal encoding performance, Fisch's can provide either, but not both, depending on its parameterization. Neither construction strictly improves on the other, however, due to the size of graph-sampling proofs of correct encoding. When parameterized for fast encoding and lower compression resistance, the Fisch's proofs grow linearly with the file size. For both our construction and the optimal-compression parameterization of the Fisch construction, the proof size is $O(n \log n)$. Our chunking optimization in Section 5.3 may be able to reduce the overall proof size for particularly large files in both constructions.

Finally, all work discussed above provides only constructions without implementations, performance evaluations, or discussions of practical ramifications.

Depth-Robust Graphs. DRGs were studied decades ago for proving lower bounds on computational complexity [26, 47, 48], and revived to enforce sequential work in publicly verifiable proofs of work [38] and memory-hard hash functions [2–4, 13], prompting several new DRG constructions [2, 38].

Publicly Verifiable Proofs of Sequential Work and Delay Functions. Publicly verifiable proofs of sequential work, e.g., Cohen and Pietrzak [19] and Mahmoody et al. [38], which uses DRGs, are public-coin systems for a prover to show 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 for applications without data recovery (e.g., timestamping, CPU benchmarks). Thus they do not support decoding, as needed for proofs of replication.

Delay functions, introduced by Goldschlag and Stubblebine [33], are similar, but do enforce uniquely computable outputs and can be decodable. Boneh et al. [12] show how to construct decodable verifiable delay functions (VDFs) that are usable in principle for proofs of replication, with fast verification (again, polylog in the prover's time) using incremental application of SNARKs—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).

11 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 allowing DSNs to ensure correct storage through a combination of verification and economic incentives. 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.

Several interesting challenges arise from our work, such as efficient verification of correct PIE computation while retaining fast data extraction and avoiding heavyweight proof systems, and support for *dynamic* provable replication, that is, files that change over time. The latter is trivially achievable by updating modified file chunks, but impractical for sparsely distributed file changes, as shown by Pietrzak [41].

In summary, 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 applications such as robust blockchain data storage, while raising questions about the cost-efficiency of others.

Acknowledgements

Many people helped with this work. First, we would like to thank our anonymous reviewers and our shepherd Jeremiah Blocki for their insightful comments and helpful suggestions. 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 W911NF16-1-0145, IC3 industry partners, and Protocol Labs (through a project-specific grant). 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/, 2019. Accessed May 2019.

[6] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/, 2019. Accessed May 2019.

[7] 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.

[8] 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.

[9] 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.

[10] J. Benet, D. Dalrymple, and N. Greco. Proof of replication. Technical report, Protocol Labs, July 27, 2017. Accessed May 2019.

[11] 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.

[12] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *International Cryptology Conference (CRYPTO)*, pages 757–788. Springer, 2018.

[13] 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.

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

[15] 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.

[16] V. Buterin. The stateless client concept. Oct. 24, 2017. Accessed May 2019.

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

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

[19] 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.

[20] 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.

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

[22] I. Damgård, C. Ganesh, and C. Orlandi. Proofs of replicated storage without timing assumptions. *IACR Cryptology ePrint Archive*, July 2018.

[23] Data storage supply and demand worldwide, from 2009 to 2020 (in exabytes). `https://www.statista.com/statistics/751749/worldwide-data-storage-capacity-and-demand/`, 2016. Accessed May 2019.

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

[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 full node sync default chart. `https://etherscan.io/chartsync/chaindefault`, 2019. Accessed May 2019.

[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.

[30] B. Fisch. Tight proofs of space and replication. *IACR Cryptology ePrint Archive*, Aug. 2018.

[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. Accessed May 2019.

[33] 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.

[34] 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.

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

[36] A. K. Lenstra and B. Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptology ePrint Archive*, Apr. 2015.

[37] S. D. Lerner. Proof of unique blockchain storage. https://bitslog.wordpress.com/2014/11/03/proof-of-local-blockchain-storage/, 2014. Accessed May 2019.

[38] 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.

[39] 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.

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

[41] K. Pietrzak. Proofs of catalytic space. *IACR Cryptology ePrint Archive*, Feb. 2018.

[42] 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.

[43] Protocol Labs. Filecoin: A decentralized storage network. July 19, 2017. Accessed May 2019.

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

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

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

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

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

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

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

[51] I. Storj Labs. Storj: A decentralized cloud storage network framework. https://storj.io/storjv3.pdf, Oct. 30, 2018. Accessed May 2019.

[52] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains. 2017.

[53] 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.

[54] 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.

[55] D. Vorick and L. Champine. Sia: Simple decentralized storage. https://sia.tech/sia.pdf, Nov. 29, 2014. Accessed May 2019.

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 compression factor $\epsilon < 1$, any polynomial number of queries s, and any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $negl$ such that for all sets of files $\{F_i\}_{i=1}^m$,

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

where

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

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

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

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

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 prove the other direction 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 $\{(F_i, \rho_i)\}_{i=1}^m$ 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 [41] 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

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

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 the 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 demonstrate our construction's security, we show that the parallel pebbling game on $\mathrm{DataEdgeGraph}(\mathcal{G})$ is $(\delta n, \beta n, \delta)$-hard for every $\delta < 1$, where $\mathcal{G}$ is the n-encoder graph in our construction. 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 (Formal)**.** *If $\mathcal{G}$ is an n-encoder graph and the parallel pebbling game on $\mathrm{DataEdgeGraph}(\mathcal{G})$ is $(\delta n, d, \delta)$-hard for every $\delta < 1$ then $\textsc{Encode}_{\mathcal{G}}$ (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 $\textsc{Encode}_{\widehat{\mathcal{G}}}$ 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 $v \in D_i$ 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 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. *Every vertex in P contains an unpebbled path of only data edges to D_ℓ, and*
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 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. By the definition of U_i, every vertex in U_i contains an unpebbled path of only data edges to an unpebbled vertex in D_ℓ, 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, and*
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 $D_\ell \setminus O$. 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 a path P' satisfying Conditions 1 and 2. By Condition 1, the terminal vertex of P' has an unpebbled path of data edges to some unpebbled $u \in D_\ell$, so we can concatenate those paths to acquire P. Because all vertices in D_ℓ outside O are pebbled in this configuration, $u \in O$.

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 $|S| \geq 2n\epsilon$. Each vertex in S has the desired property, proving the claim. □

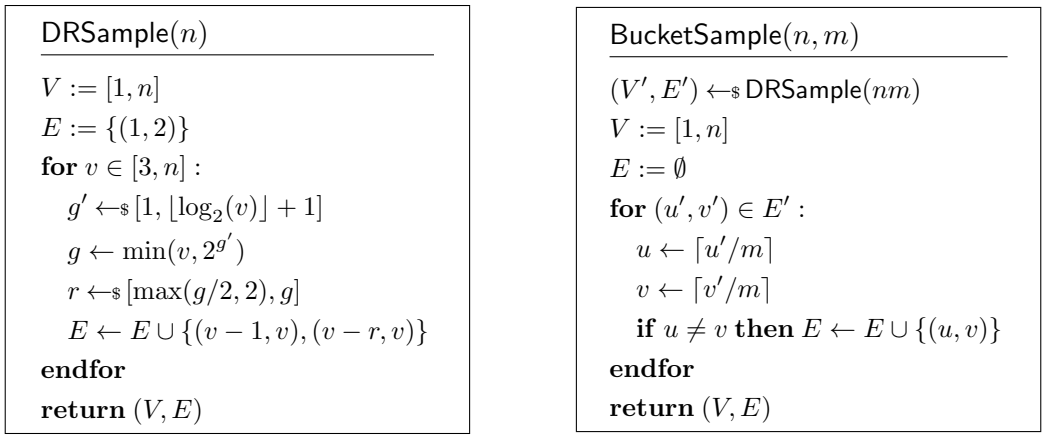


Figure 9: The DRSample algoirthm for sampling degree-2 DRGs due to Alwen et al. [2], and the modification into BucketSample due to Fisch et al. [32]. These algorithms produce very simple low-degree graphs that appear depth-robust in practice.

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 DRG Construction Details

We now describe the exact construction details for both DRGs we describe in Section 7 and benchmark in Section 8.

C.1 Naïve DRG

Constructing a naïve (α, α)-DRG with in-degree $(1 - \alpha)n$ is straightforward. We index the 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. The graph is therefore (α, α)-depth-robust as desired, but it has in-degree $(1 - \alpha)n$.

C.2 Random Sampled Heuristic DRG

Alwen et al. [2] present an algorithm for randomly sampling degree-2 graphs that appear fairly depth-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 and depth-robustness that grows with in-degree. The sampling algorithms for both are shown in Figure 9. For sampled graphs with n vertices and maximum in-degree 21, Fisch was 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 $(0.5, 0.25)$-depth-robust.