PIEs: Public Incompressible Encodings for Decentralized Storage

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

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

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

†Initiative for CryptoCurrencies & Contracts

Abstract

We present the first provably secure, practical approach to proving file replication (or other erasure coding) in distributed storage networks (DSNs). Storing multiple copies of a file F is essential in DSNs to ensure against file loss in the event of faulty servers or corrupt data. The *public* nature of DSNs, however, makes this goal challenging: no secret keys or trusted entities are available. Files must therefore be encoded and decoded using public coins—i.e., without encryption or other secret-key operations—and retention of files by servers in the network must be publicly verifiable.

We introduce and formalize the notion of a *a public incompressible encoding (PIE)*, a tool that allows for file-replication proofs in this public setting. A PIE enables public verification that a server is (nearly) entirely storing a replicated encoding G of a target file F, and has not deduplicated or otherwise compressed G to save storage. In a DSN with monetary rewards or penalties, a PIE helps ensure that an economically rational server is incentivized to store G and thus replicate F honestly.

We present a specific PIE based on a novel graph construction, called a *Dagwood Sandwich Graph* (DSaG), that includes long paths even when an adversary selectively discards edges. This PIE ensures that a cheating server must perform a large (and completely tunable) number of costly *sequential* cryptographic operations to recover any blocks of G it chooses to discard. By periodically challenging the server to return randomly selected blocks of G and timing the responses, the DSN can thus verify that a server is storing G intact.

We prove the security of our PIE construction and present performance evaluations demonstrating that it is efficient in practice—empirically within a factor of 6.2 of optimal by one metric. Our proposed PIE offers a valuable basic tool for building DSNs, such as the proposed Filecoin system, as well as for other challenging file-storage needs in public settings. PIEs also meet the critical security requirements for such applications: they preclude demonstrated attacks involving parallelism and acceleration via ASICs and other custom hardware.

1 Introduction

The world's data storage demands are ballooning, with annual growth rates of 42% and a projected 50 zettabytes required by 2020 [35]. Supply, however, is lagging [18], 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 [42], Storj [43], MaidSafe, IPFS, and Filecoin [34] that store data on unused devices in peer-to-peer systems. DSNs meet not only a universal demand for storage, but also specific needs in blockchain systems. Ethereum's heavily replicated blockchain, for example, recently grew from 20 GB to over 75 GB in less than a year, prompting calls for new decentralized storage architectures [11].

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) [29, 38] and Proofs of Data Possession (PDPs) [4] allow an untrusted provider to prove retention of a file F efficiently to a client. Unfortunately, these techniques do not enable a client or verifier to distinguish between an honest provider that ensures robustness by storing three copies of F and a cheating provider that stores a single, brittle copy of F. Proving file replication thus requires a different set of techniques. This is particularly problematic, as proposed DSNs envision rewarding broader replication, thus enshrining direct incentives for such malfeasance in the protocol.

1.1 The Challenge of Proving Replication

It is straightforward to prove replicated storage of a file F in a restricted *trusted-encoder*, *private-reader* setting where the file owner/client can use secret keys to encode and retrieve F. To store three copies, the client simply generates secret keys $(\kappa_1, \kappa_2, \kappa_3)$, encrypts F under each key respectively, and uploads the resulting ciphertext triple $G = (C_1, C_2, C_3)$. The provider can prove to the client using a PoR/PDP on G that it is storing all copies. Use of encryption prevents the provider from cheating and discarding a file copy or compressing file contents—but it also means only the owner can retrieve F. Systems such as Sia [42] and Storj [43] support this approach.

Proving replication is also possible in a *trusted-encoder*, *public-reader* setting. Existing mechanisms [17, 41] show how an encoder (e.g., file owner) can perform a private-key operation using a trapdoor one-way function (RSA) to encode individual replicas of F. Any entity can recover F from this encoding with a private key. Additionally, anyone can verify storage of any replica using a publicly verifiable PoR/PDP.

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

The public setting is essential for several applications. In Filecoin, for instance, miners prove replicated storage of public files [34]. They have an incentive to cheat in order to reduce their storage costs, and thus are untrusted encoders. Similarly, smart contracts cannot manage secret keys, so they must store data via untrusted encoders.

Proving replicated file storage in the public setting is technically challenging, though. To begin with, any proof of storage protocol for 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 $G = \text{ENCODE}(F)$ as needed. Constructing a good function ENCODE that imposes a strong lower bound on a cheating prover's response time is thus a major technical challenge.

As we explain in Section 2.2, previous attempts to construct a good encoder ENCODE have either contained serious security flaws [6] or required implausible assumptions [41].

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

1.2 A Practical Public Incompressible Encoding (PIE)

We introduce the first practical, provably secure technical tool to enable proofs of replication in the public setting: a *public incompressible encoding* (PIE).

Informally, a PIE 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 by a verifier to produce a randomly selected block of G will with probability at least ϵ (minus a negligible term) need to perform a long sequential computation. The resulting delayed response will be detectable by the verifier. Soundness can be boosted in the standard way with multiple queries.

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

We emphasize that PIEs have other important applications. They can help minimize storage in permissioned blockchains via verifiable erasure coding across nodes. In cloud systems, they can help enforce encodings other than file redundancy, e.g., at-rest encryption, watermarking of files, and binding of licenses to data, as proposed in [41]. There are no doubt many other applications as well.

Our Construction. Our proposed PIE ENCODE is a graph-based file transformation. Specifically, it depends on a new construct that we call a *Dagwood Sandwich Graph* (DSaG).[2] A DSaG is an iterated interleaving of two graph components. The first is a *depth-robust graph* (DRG)—intuitively, a directed acyclic graph with the property that even if an adversary removes a large number of nodes, a long sequential path remains. Depth-robust graphs are commonly used in memory-hard hash function designs [1–3, 14] and have been used to enforce long sequential computations [31], a purpose we exploit in our DSaG construction. The second component is a classic butterfly graph [16], also used in the (weak) PIE construction by van Dijk et al. [41]. Intuitively, a butterfly graph creates a dependency of every output node on all input nodes, while ruling out possible "shortcuts." By sandwiching the two graphs and iterating, we can prove that any attempt at compression, such as discarding a block of G and attempting to recompute it on the fly, forces a provider to recompute blocks of G in an expensively slow and detectable fashion.

We show experimentally that, for a variety of realistic and highly tunable concrete timing bounds, our construction's performance is within a factor of 6.2 of the theoretical optimum by one metric. Moreover, our construction operates on files as small as tens of kilobytes and lends itself to high parallelism, allowing it to scale very efficiently to larger files.

Paper Organization and Contributions

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

- *Public Incompressible Encodings*: We introduce and formally define the notion of a *public incompressible encoding* (PIE) in Section 3.
- *Dagwood Sandwich Graphs:* In Section 4 we define *encoder graphs* and prove properties needed for secure PIEs. In Section 5, we present and prove security for an encoder graph called a *Dagwood Sandwich Graph* (DSaG), a novel construction involving an iterated interleaving of depth-robust graphs with butterfly graphs.

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

- *Implementation and experiments:* We offer performance optimizations in Section 6 and provide an efficiency metric and numerical parameters for strong security in Section 7. In Section 8 we report experimental performance results that validate the practicality and near optimality of our approach.

We discuss related work in Section 9 and conclude in Section 10.

2 Background

By way of background, we now give a brief overview of how PIEs may be used in DSNs and the challenges in their construction illustrated by previous work.

2.1 Using Incompressible Encodings in DSNs

Incompressible encodings do not, by themselves, solve the problem of ensuring file storage robustness in DSNs. They do not spread storage across multiple nodes or ensure data availability. They do, however, provide a necessary component for building any such system by enabling *detection of data-replication failures*.

Recall that DSNs aim to ensure that a file F is stored redundantly, i.e., with erasure coding or replication (a simple form of erasure coding). The idea is that if some copies or pieces of F are lost or corrupted, F should still be recoverable.

In a public DSN, however, storage servers are considered potentially untrustworthy. To encourage good behavior, they are periodically compensated for storing data. As a result, they have an incentive to reduce storage costs as much as possible: the more data a server can 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 [4, 29, 38], as noted above, can detect when a server fails to store a target file F. But they cannot detect failures to store a file F redundantly—with requested replication.

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

An alternative method of detecting deduplication is to *time* the responses of servers when challenged to prove possession of F. If S_2 must obtain F from S_1, it will respond to challenges slower than S_1, which can access F directly. Such timing, though, can be highly unreliable given variance in network latency.

Public incompressible encodings (PIEs), our focus here, address this problem in the public setting described above by detecting failures to store F redundantly. They rely on timing assumptions, but use a slow file transformation that makes server response times tunable, rendering timing of server responses robust to variable network latency. PIEs can also verify file replication across multiple servers or simply on a single server, if desired.

The general procedure for applying a PIE is as follows. The target file F is first transformed into a redundant state via duplication, block-level erasure coding, or something else. For example, F could be replicated in triplicate as $(F_1 \,\|\, F_2 \,\|\, F_3)$, where $F_i = F$. This redundant version of F is then encoded using the PIE into a representation G, e.g., $G = (G_1 \,\|\, G_2 \,\|\, G_3)$ for $G_i = \textsc{Encode}(F_i)$. Of course, the DSN needs to ensure that G is correctly encoded. We briefly discuss below how this might be achieved.

Once G is stored, DSN servers are challenged periodically to prove possession of G. An incompressible encoding ENCODE guarantees that if G is in any way compressed—some representation of G is not stored

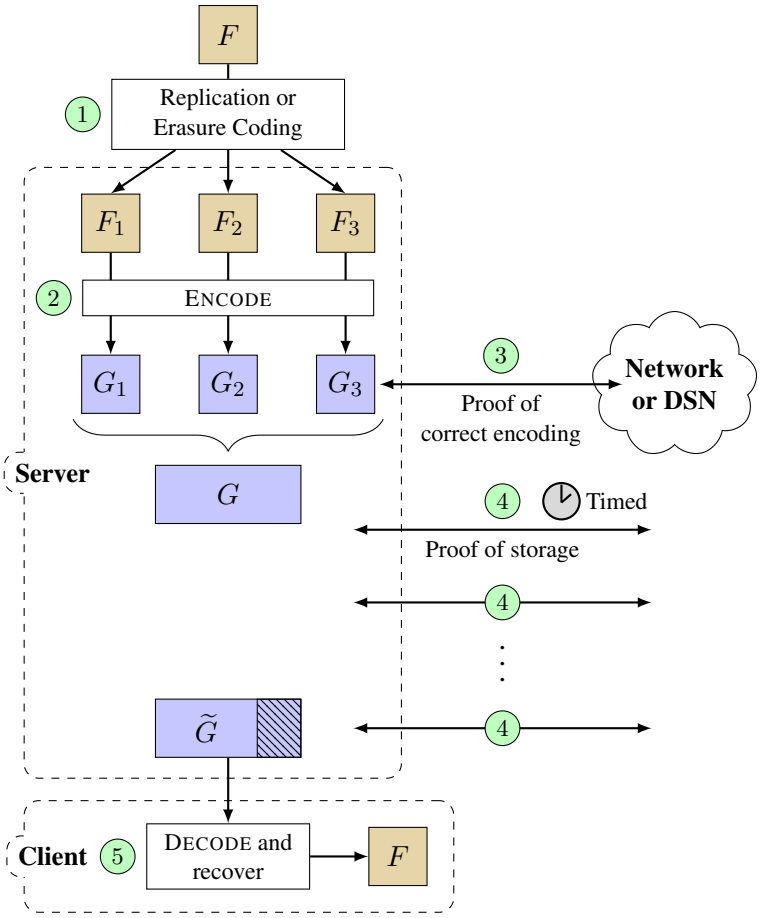


Figure 1: The basic structure of a DSN using a PIE to ensure data robustness. (1) First a file F is replicated or erasured coded to ensure robustness. (2) The server runs this robust data through ENCODE, preventing compression. (3) The network/DSN queries the server to prove the data was encoded properly.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

in its entirety—then it will be possible to detect this fact. If G is fully stored, then F is still retrievable even given damage to G. For example, if G_1 and G_2 are lost, G_3 can be decoded to recover F. Consequently, with this arrangement a DSN can verify the erasure-resistance of its data.

We show the steps for application of a PIE in a DSN in Figure 1. Application of PIEs for other goals, such as proving that files bear watermarks [41], look similar.

Should a server be caught *not* storing G, it can be financially penalized by not receive a reward or being forced to forfeit funds placed in escrow. (See, e.g., [34].) Storing most but not all of G would incur a substantial risk of penalty in addition to ongoing storage costs, while storing little of G would mean an overwhelming probability of detection and penalization. Given suitably calibrated rewards, a rational server is thus incentivized to store G correctly and in its entirety. The problem of how to set financial penalties in a DSN is out of scope for this work.

Also out of scope is the orthogonal problem of ensuring that G is *distributed across servers or storage devices*; see, e.g., [7, 10] for approaches to this problem.

Ensuring Correct PIE Computation. In the public setting we consider, no entity can be trusted to compute G correctly. A PIE itself does not ensure correct encoding—only incompressibility. Thus another challenge arises: ensuring that G is correctly computed from F, i.e., implementing Step 3 from Figure 1 above.

In one sense this problem is trivial. If anyone can encode or decode the data, then every node in a DSN can simply check than $G = \text{ENCODE}(F)$. Unfortunately, use of timing assumptions inherent in PIE construction requires encoding to be slow and resource-intensive. As a result, having every node in the DSN validate correct encoding would result in prohibitively high computational costs for the network.

There are multiple ways to address this concern, with the most appropriate approach depending upon the setting. For example, in a permissioned setting or hybrid consensus system [32], which selects small committees presumed to contain a quorum of honest nodes, a subset of nodes can verify correct file encoding.

In a permissionless settings, an alternative approach is not to verify correct encodings, but to creating financial incentives to deter misbehavior. There are two ways to do this. We briefly sketch a "positive" scheme where the encoder produces proof that an encoding is *correct*; this scheme detects misbehavior with high but not overwhelming probability. We also describe a "negative" scheme in which any entity can generate a compact proof that shows with overwhelming probability that an encoding is *incorrect*. These approaches are complementary and further improve security when taken together.

The "positive" proof approach relies on the fact that the ENCODE function in our PIE scheme is defined by a graph. It is possible—non-interactively when encoding takes place—to randomly sample vertices in this graph and prove that they were computed correctly with respect to their parents. Using this approach, an encoder can prove that a large percentage of the vertices were correctly computed. A small number of incorrect vertices could escape detection, but given a suitable penalty, the need to provide a positive proof could deter a rational encoder from cheating.

The "negative" proof approach relies on our observation that anyone can recompute ENCODE or DECODE and thus check that $G = \text{ENCODE}(F)$. It is possible then to construct a compact proof that shows with overwhelming probability that a particular encoding is *incorrect*. This proof can be constructed via non-interactive verifiable computation [25], or interactively using more straightforward binary search techniques and refereed delegation of computation [5, 12], such as those popularized by Truebit for detecting incorrect smart contract computation [40]. With this approach, it is possible for the network to financially reward whistleblowing, incentivizing users to verify encodings and submit proofs when they are incorrect. The flip side is that it is then possible to penalize incorrect computation by imposing financial penalties for encodings proven to be incorrect.

As the right mechanism for verifying correct encoding is context-dependent and technically orthogonal to PIE construction and use, and due to space constraints, we omit details on these various approaches.

2.2 Previous Approaches to PIE Construction

To understand the challenges involved in constructin a PIE, we briefly describe two previous works that implicitly attempt to provide incompressibility.

Filecoin proposed a PIE, called a *proof of replication* (PoRep), specifically to prove file replication [6, 34]. The Filecoin PIE computes an encoding $G = (G_1 \,\|\, G_2 \,\|\, G_3)$ by computing $G_i = \text{ENCODE}_i(F_i)$ where $F_i = F$. The function ENCODE_i involves multiple, chained (CBC) encryption passes over all of F_i under a published key κ_i, making every output block of G_i dependent on every input block of F. Consequently, it would seem that if a server discards a block from G_i, it must redo all of the work of ENCODE_i to recompute

it, resulting in a detectably slow response to a challenge.

Unfortunately, previous work [41] anticipates an attack against this scheme. A cheating provider can checkpoint specific intermediate states of ENCODE_i as "shortcuts." It can use these values to deduplicate G (i.e., compress $G_1 \parallel G_2 \parallel G_3$), and, when challenged, quickly recompute any missing challenge blocks.

An alternative PIE proposed in [41] is provably resistant to such "shortcuts." Called an *hourglass function*, it applies a published-key pseudorandom permutation (PRP) to pairs of blocks of F in a sequence determined by a butterfly network. Hourglass functions and related approaches (e.g., [10]), however, assume slow retrieval of F due to use of rotational hard drives. The proliferation of fast storage like solid-state drives (SSDs) and monetary incentives to cheat in systems like Filecoin clearly invalidate this assumption. We use butterfly networks, however, as a component of our PIE construction.

Consequently, previous work offers no general, practical approach to the problem of remotely proving storage redundancy for publicly readable files F.

3 Security Definitions

Here we define a secure *public incompressible encoding* (PIE) scheme.

An *encoding* is a function pair $(\text{ENCODE}, \text{DECODE})$ such that for all files F and coins ρ,

$$\Pr[(G, \kappa) \leftarrow \text{ENCODE}(F\,;\,\rho) : \text{DECODE}(G, \kappa) = F] = 1.$$

Here, κ is a key output by ENCODE and stored with F for use by DECODE. An encoding is *public* if the coins ρ are public.

Informally, we define an encoding as *incompressible* as follows. An adversary $\mathcal{A}$ chooses a set of files $\{F_i\}$, which may or may not be identical copies of some file F. When the $\{F_i\}$ are encoded to $G = \{G_i\}$ under random coins, $\mathcal{A}$ is challenged to produce randomly selected blocks of G. The encoding is incompressible if $\mathcal{A}$ can only respond successfully to challenges (except with negligible probability) by using dedicated storage at least equal to $|G|$. In other words, $\mathcal{A}$ cannot compress or deduplicate G. Without any special assumptions in place, such incompressibility is impossible to achieve in the public setting. $\mathcal{A}$ can cheat simply by choosing $F = F_1 = F_2$, storing F, and responding to challenges by computing blocks of G (for the appropriate G_i) on the fly. Thus $\mathcal{A}$ is storing only $\frac{1}{2}|G|$ blocks, meaning it has achieved a factor of two compression. In fact, for any k, $\mathcal{A}$ can achieve a factor of k compression by choosing k copies of F.

Our goal is to make such cheating by $\mathcal{A}$ detectable. We do this in practice by ensuring that computing ENCODE is time consuming. A cheating adversary, then, will incur a detectably long delay before being able to respond to challenges.

Modeling this notion of time-consuming computation, however, is tricky. As conventional security definitions typically consider only the overall computational costs of an attacker, they often rely on fragile computational models that treat heterogeneous operations uniformly. We take a different approach. We only limit the number of times an attacker can execute a specific operation: a key derivation function that we can make intentionally slow.

Specifically, our construction relies on a key derivation function which we model as a random oracle $\mathcal{O}$. We then constrain the number of calls to $\mathcal{O}$ that an adversary can make while responding to queries. As a real adversary may be able to perform computation in parallel, we only constrain the number of *sequential* calls to $\mathcal{O}$, not the overall number. To model this, we define a parallel oracle $\mathcal{O}_d^\star$ in terms of $\mathcal{O}$ that will respond to any number of simultaneous queries, but only $d - 1$ sequential ones. $\mathcal{O}_d^\star$ is initialized with a counter $c = 1$ and is defined as

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

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

By having ENCODE require sequential calls to the oracle and allowing only $d-1$ such calls, we can excluded attackers who could be caught be timing measures meant to prevent on-the-fly re-encoding attacks. Using this tool we now explore what it means for an encoding to be incompressible.

Recall that our goal, informally, is to ensure that an adversary actually stores as much data as they claim to store. This allows us to store redundant publicly retrievable data on potentially-colluding servers or fairly pay a provider for storage used, even when that provider may have maliciously constructed the files it is storing. This includes the form of cheating involving on-the-fly re-encoding described above, but also more subtle forms of cheating. $\mathcal{A}$ need not cheat by computing ENCODE directly. It can store arbitrary values, including intermediate values in the computation of ENCODE, such as the "shortcuts" used to attack Filecoin's protocol. Additionally, the adversary can use knowledge of the keys $\{\kappa_i\}$ to cheat by creating multiple files that encode to the same thing—e.g., by encoding F_1 with κ_1 to get G_1 and decoding G_1 with κ_2 to get F_2—and more. Our security definition addresses all of these possible adversarial strategies.

We thus define security in terms of an adversary who must return encoded file blocks given limited storage. For a file G, we use $|G|$ to denote the length of G in units of λ-bit blocks and $G[i]$ to mean the ith such block. For a random oracle $\mathcal{O}$, we define $\mathcal{O}_d^\star$ as above.

Definition 1 (Public incompressible encoding)**.** We say a public encoding algorithm ENCODE is *d-oracle incompressible* if for any number of files $m \geq 1$, any compression factor ϵ, and any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $negl$ such that

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

where

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

$\{\rho_i\}_{i=1}^m \leftarrow\!\!\$\, \{0,1\}^{\lambda \times m}$
$(\{F_i\}_{i=1}^m, G') \leftarrow\!\!\$\, \mathcal{A}_1^{\mathcal{O}}(\{\rho_i\}_{i=1}^m)$
for $i \in [1, m] : (G_i, \kappa_i) \leftarrow \text{ENCODE}^{\mathcal{O}}(F_i\,;\, \rho_i)$
$G = G_1 \,\|\, \cdots \,\|\, G_m$
$j \leftarrow\!\!\$\, [1, |G|]$
$blk \leftarrow\!\!\$\, \mathcal{A}_2^{\mathcal{O}_d^\star}(j, G', \{(\rho_i, \kappa_i)\}_{i=1}^m)$
return $\left(\dfrac{|G'|}{|G|} \leq 1 - \epsilon\right) \wedge (blk = G[j])$

Definition 1 considers an adversary $\mathcal{A}$ who claims to store m honestly encoded files, but attempts to compress the combined encodings by a factor of ϵ. We consider a PIE scheme to be secure if, when randomly challenged to return one encoded block, $\mathcal{A}$ can do no better than simply discarding an ϵ fraction of the encoding blocks and hoping the challenge is not in the discarded set. This rules out the storage "shortcuts" used to attack schemes like the ones in Filecoin.

By only bounding $\mathcal{A}_2$ by the number of inherently sequential queries it makes to $\mathcal{O}$, we avoid relying on questionable timing assumptions about rotational hard drive latency [41] or, indeed, any other explicit timing

bounds. Importantly, our definition allows $\mathcal{A}_2$ to model an adversary with unbounded parallelism. This property is highly desirable given current developments in so-called ASIC mining [39], which uses custom hardware to achieve modest improvements in sequential performance and drastically increased parallelism.

We view the challenge process as modeling 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—a common assumption frequently approximated in practice using memory-hard hash functions. In contract, we place only standard polynomial time bounds on $\mathcal{A}_1$, as it represents an attacker's offline attempt to identify compressible files.

Parallel and Sequential Composition. We note that Definition 1 is secure under parallel composition because of its use of multiple files and honestly-chosen randomness. Indeed, multiple parallel executions of $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}$ can be compressed into one by summing the number of files. As $\mathcal{A}_1$ is presented with all randomness at the same time, it can achieve the same correlations and collisions within one larger game as it could across multiple smaller ones.

Separately, one could consider the sequential composition of multiple challenges and the potential for some sort of amortization. In Appendix A we consider a definition that allows multiple such challenges and argue that it is equivalent. Informally, for any sequence of challenges, the multi-challenge adversary must realize a storage advantage on at least one challenge. Thus, given an adversary $\hat{\mathcal{A}}$ for a multi-challenge definition who first gains an advantage on challenge i, we can construct an adversary $(\mathcal{A}_1, \mathcal{A}_2)$ for Definition 1 where $\mathcal{A}_1$ simulates $\hat{\mathcal{A}}$ up to query i, and $\mathcal{A}_2$ behaves on its single challenge as $\hat{\mathcal{A}}$ would on the ith query.

4 Encoder Graphs

Now that we have formally defined the security of PIEs, we need to construct one. Our encoding will consist of a keyed pseudorandom permutation PRP_κ applied to the blocks of the file in some sequence using keys derived by some key derivation function KDF, which we model as a random oracle $\mathcal{O}$. It is calls to $\mathcal{O}$ that will constitute the slow, sequential computation path in our PIE construction. Intuitively, if the encoding requires a large number d of sequential key derivations, an attacker will be caught if it attempts to dynamically recompute the encoding.

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

To reduce the overall cost of encoding and decoding while retaining the same security, we consider alternate ways of sequencing KDF and PRP operations. In particular, we aim to minimize the aggregate cost of encoding (or decoding) the entire file, while maintaining a high cost to encoding individual blocks. We stress that this must apply for *any* block, as otherwise an adversary could discard a block and rapidly recompute it on the fly, thus making the encoding compressible. The question is how to structure the encoding function that interleaves PRP and $\mathcal{O}$ operations to achieve this.

We analyze ENCODE and DECODE by treating them as circuits with n input wires and n output wires corresponding to the blocks in the file. (The encoding key is taken as an auxiliary input.) We model the internal structure of the circuit as a directed acyclic graph (DAG) where each vertex represents both a call to $\mathcal{O}$ and a PRP operation. As such, the graph has two types of edges: data edges and key edges. Data edges represent lossless flows of data—the operation performed on blocks passed along data edges must transform the data in an invertible fashion. Key edges represent (lossy) dependencies used to determine how other data

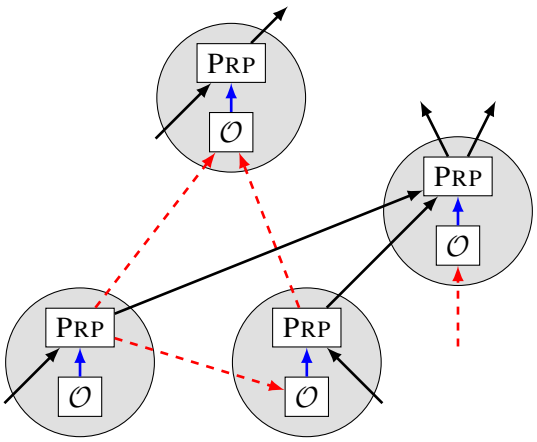


Figure 2: The high level structure of an encoding based on a DAG. Solid black edges are data edges, dashed red edges are key edges, and blue arrows from $\mathcal{O}$ to PRP show the use of derived keys.

is transformed. Therefore, as depicted in Figure 2, at each node we call $\mathcal{O}$ on all data passed in along key edges, and use the resulting key to permute data passed in along data edges using PRP.

4.1 Ensuring Decodability

While we model our encoding function as a DAG, not all DAGs represent valid encodings—some cannot be decoded. To this end we define n-encoder graphs, which specify a means of encoding n block files with no expansion in file size.

In order to undo the operation at a given vertex, one needs the value along all outgoing data edges and the values along all incoming key edges. We can thus check if the transform defined by a DAG is invertible by looking for cycles in a graph representing these inverse dependencies. A cycle in the decoder graph would indicate some intermediate state that is necessary for decoding and not computable from only the original encoding's output.

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

Definition 2 (Encoder graphs)**.** Let $\mathcal{G} = (V, E)$ be a DAG where E can be partitioned into data edges E_D and key edges E_K. For a vertex $v \in V$, we use $\text{DEG}^D(v)$ to 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 (in-degree 0) and n sink nodes (out-degree 0).
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 ($\text{DEG}^K(v) = 0$),
   b) v produces the same amount of data as it consumes ($\text{DEG}^D_{\text{in}}(v) = \text{DEG}^D_{\text{out}}(v)$).
3. For all key edges $(u, v) \in E_K$, the origin vertex produces one data block—$\text{DEG}^D_{\text{out}}(u) = 1$.

4. $\mathcal{G}$ represents an invertible transform. Specifically, there are no cycles in $\mathcal{G}^{-1} = (V, E^{-1})$, where

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

Condition 4 formalizes the intuition above about decoding. To invert the computation at a vertex v, we need the values originally output along all data edges, so those edges simply reverse direction. We also need the value of each incoming key edge (w, v). The values produced by w during encoding, however, are now produced by the vertex u where w's outgoing data edge terminates. Thus, to properly model the data dependency for decoding, we replace (w, v) with (u, v). As noted above, $\mathcal{G}$ represents an invertible transformation if this new graph $\mathcal{G}^{-1}$ has no cycles. None of our security theorems consider decoding explicitly, so we do not consider $\mathcal{G}^{-1}$ again.

Note that Definition 2 is far from totally general. It does not allow the encoding to condition on data or modify the file size at all. Condition 3 requires that data be separated into single blocks before being used in lossy dependencies. We conjecture that all results in this work hold without this restriction, but it drastically simplifies definitions and proofs.

4.2 Compression-Robustness

We can now analyze which encoding graph structures are robust to compression attempts. The incompressibility of an encoding defined by an n-encoder graph $\mathcal{G}$ depends on whether or not an adversary $\mathcal{A}$ can store intermediate state in lieu of performing computation.

As $\mathcal{G}$ defines the computation of our original encoding, we can modify $\mathcal{G}$ based on the data $\mathcal{A}$ is storing to represent the computation that $\mathcal{A}$ must redo in order to compute output blocks and respond to challenges. Recall that we represent data as edges and computational operations as vertices in $\mathcal{G}$. An edge (u, v) means a block of data output by u is required to compute v. If $\mathcal{A}$ is storing the data associated with (u, v), it has removed the computational dependency since there is no longer any need to recompute u before computing v. We thus model this dependency elimination by removing (u, v) from $\mathcal{G}$. As each edge represents a single block of data, we can represent $\mathcal{A}$ storing k arbitrary blocks by removing k edges.

This simple model unfortunately ignores the distinction between data edges and key edges. As described, it works perfectly for data edges, but a single vertex may have arbitrarily many key edges and each might output the same value. In this case $\mathcal{A}$ could remove the dependencies of all of these key edges by storing a single block. We leverage the restriction that key edges must originate at vertices with only one data edge to conservatively approximate $\mathcal{A}$'s storage power: if $\mathcal{A}$ stores the block represented by a data edge (u, v), we also remove all key edges originating at u. We then remove one *data edge* for each block $\mathcal{A}$ stores and any key edges originating at the same vertices. This way, even if all key edges have the same value as the one data edge—which they do in our construction—we still properly remove any destroyed dependencies. We denote the sub-graph with this reduced edge set as $\mathcal{G}'$.

To ensure security under Definition 1 we must ensure that if $\mathcal{A}$ discards an ϵ fraction of the output blocks in the encoded file G, it must still make at least d sequential calls to $\mathcal{O}$ when recomputing those ϵn discarded blocks. As we model the remaining computational dependencies with our modified $\mathcal{G}'$, this corresponds to $\mathcal{G}'$ retaining long paths to an ϵ fraction of output nodes if the amount of data stored—and thus the number of data edges removed—is at most $(1 - \epsilon)n$. The length of a path in $\mathcal{G}'$ to an output vertex places a lower bound on the number of sequential operations needed to compute that output. For data edges, it is possible

the corresponding key can be computed in parallel, so to ensure at least d sequential calls to $\mathcal{O}$, the path must contain at least d key edges.

Normalizing to $\delta = \frac{d}{n}$, this leads to our formal definition of compression-robustness.

Definition 3 (Compression-Robust Graph)**.** Let $\mathcal{G} = (V, E)$ be an n-encoder graph with data edges E_D and key edges E_K. For any set of data edges $\tilde{E}_D \subset E_D$, let

$$\tilde{E}_K = \{(u, v) \in E_K \mid \exists (u, w) \in \tilde{E}_D\}$$

be the set of key edges whose origin vertices also originate an edge in $\tilde{E}_D$. Now let $E' = E \setminus (\tilde{E}_D \cup \tilde{E}_K)$.

We say $\mathcal{G}$ is (γ, δ)-*compression-robust* if $\frac{|\tilde{E}_D|}{n} < \gamma$ implies that there exists a path $P \subseteq E'$ in this reduced edge set that ends in a sink node of $\mathcal{G}$ and contains at least δn key edges.

This notion is very similar—and in fact inspired by—that of a *depth-robust graph* (DRG), though DRGs consider vertex removal.[3] Originally due to Erdös, Graham, and Szemerédi [22], a DRG is a graph that retains high depth even when many vertices are removed.

Definition 4 (Depth-Robust Graph)**.** A graph $\mathcal{G} = (V, E)$ is (γ, δ)-*depth robust* if for all $\tilde{V} \subset V$ with $\frac{|\tilde{V}|}{|V|} < \gamma$, the induced subgraph on $V \setminus \tilde{V}$ has a path of length at least $\delta|V|$.

DRGs are useful for constructing memory-hard functions [1–3, 14] and have been used by Mahmoody et al. [31] to ensure inherently sequential work. As we describe in Section 5, we use them for a very similar purpose to construct compression-robust graphs.

A (γ, δ)-compression-robust graph only ensures long paths if we remove fewer than γn data edges, but an incompressible encoding must detect *any* discarded data, so we need $\gamma = 1$. This would make little sense in a DRG. A $(1, \delta)$-DRG with n vertices is impossible for $\delta > 0$, as $\gamma = 1$ allows removal of every vertex, and hence every path. In compression-robustness, however, γ and δ are defined as fractions of the number of output nodes, allowing the entire graph to be considerably larger. This flexibility makes $(1, \delta)$-compression-robust graphs possible for nontrivial δ. As we show in Section 5, we can construct an n-encoder graph $\mathcal{G}$ that is $(1, \delta)$-compression-robust with $|\mathcal{G}| = O(n \log n)$ and $\delta = \Omega(1)$.

Lastly, if an adversary discards an ϵ fraction of the output blocks in G—corresponding to storing $|\tilde{E}_D| = (1 - \epsilon)n$ data blocks—Definition 3 only guarantees a single long path. Our security definition, however, requires one for each of the ϵn discarded output blocks. Thankfully, a simple inductive argument shows that if $\frac{|\tilde{E}_D|}{n} < \gamma - \epsilon$, there exist ϵn paths each containing δn key edges and terminating in a different output node. We formalize this claim as Proposition 2 in Appendix B.

4.3 Building a PIE

We now describe how to construct a PIE from a $(1, \delta)$-compression-robust graph $\mathcal{G}$. Intuitively, at each vertex v in $\mathcal{G}$, we invoke our slow oracle $\mathcal{O}$ on all incoming key edges to derive a key κ_v. We then combine the values coming in along each data edge and permute them using a pseudorandom permutation (PRP) keyed by κ_v. The result is sent out along any outgoing key edges as well as outgoing data edges.

Formally, we let $\textsc{Kdf}^\ell : \{0,1\}^* \to \{0,1\}^{\ell\lambda}$ be a family of key derivation functions that we model as a random oracles, and $\textsc{Prp}^\ell_\kappa : \{0,1\}^{\ell\lambda} \to \{0,1\}^{\ell\lambda}$ be a family of keyed PRPs that we model as ideal ciphers. Using this notation, Construction 1 specifies how to construct a δn-oracle PIE from a $(1, \delta)$-compression-robust n-encoder graph.

[3] Some previous defines depth-robustness in terms of edge removal [36, 37]. This is more similar to our definition, but less useful for our construction.

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

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

- For input vertices $v_1, \ldots, v_n$, assign $F[i]$ to the single outgoing (data) edge e_i from vertex v_i.
- Let v_i be neither a source nor sink node that is the ith vertex. Let $x_1, \ldots, x_k$ be the values assigned to each incoming key edge and $y_1, \ldots, y_\ell$ be the values assigned to each incoming data edge.

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

  For each outgoing key edge, assign y' to that edge. For outgoing data edges $e_1, \ldots, e_\ell$, assign y'_i to e_i.

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

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

Theorem 1. *If $\mathcal{G}$ is a $(1, \delta)$-compression-robust n-encoder graph, then $\textsc{Encode}_{\mathcal{G}}$ as defined in Construction 1 is a δn-oracle public incompressible encoding.*

We provide a proof of Theorem 1 in Appendix B.

5 Dagwood Sandwich Graphs

We have shown in Theorem 1 that we can construct a PIE from a $(1, \delta)$-compression-robust graph. We now must build such a graph. Our construction, which we call a *Dagwood Sandwich Graph* (DSaG), is based on layering multiple copies of a DRG each separated by a butterfly network.

To understand why this iterated construction is necessary, we first examine some simpler strawmen.

A Single DRG. We can straightforwardly construct a n-encoder graph from any 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. As the compression-robustness definition removes all key edges originating at a vertex whose sole data edge is removed and every vertex in $\mathcal{G}$ has one outgoing data edge, removing that data edge is equivalent to removing the vertex. If $\mathcal{G}$ is (γ, δ)-depth-robust, this construction straightforwardly results in a (γ, δ)-compression-robust graph.

Unfortunately, this graph cannot produce a d-oracle incompressible encoding for any nontrivial d. While some vertices terminate paths containing many key edges, others to not. In particular, since $\mathcal{G}$ is a DAG, there is at least one vertex v with no incoming key edges. An adversary $\mathcal{A}$ can simply discard the output block produced by that vertex and recompute it very rapidly if queried.

Multiple DRGs. In order to address the attack outlined above, we can attempt to layer multiple copies of $\mathcal{G}$ together. 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. Unfortunately, if $\mathcal{A}$ discards an ϵ fraction of the blocks (corresponding to removing $(1 - \epsilon)n$ data edges), it may still be able to recompute those blocks with only a small number

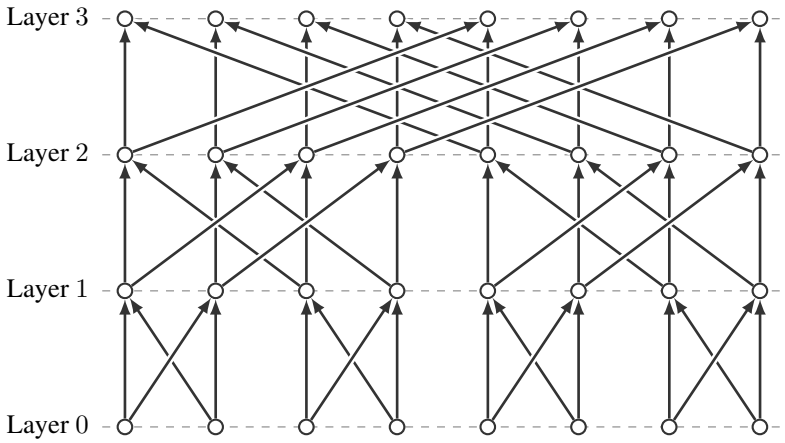


Figure 3: $\mathcal{B}_3$, a butterfly network for 8 inputs and outputs.

of oracle calls. We can guarantee that a long path exists in one of these copies of $\mathcal{G}$, but the end of that path may no longer connect to any output vertices of the encoder graph.

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

We must, however, be careful about how we perform 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—which was originally suggested as an entire encoding scheme by Filecoin [6]—appears to create the required connection. Regrettably, as shown by van Dijk et al. [41], the dependencies can be broken by storing a small number of "shortcut" blocks.

Dagwood Sandwich Graphs. There is a much more robust connector than a series of CBC passes, namely a *butterfly network* [16]. A butterfly network $\mathcal{B}_k$ is a graph with in-degree 2 that connects each of 2^k inputs to all of 2^k outputs. It does this with k layers of 2^k vertices each, where each vertex in one layer is connected to two vertices in the next. Figure 3 shows $\mathcal{B}_3$, a butterfly network on 8 elements.

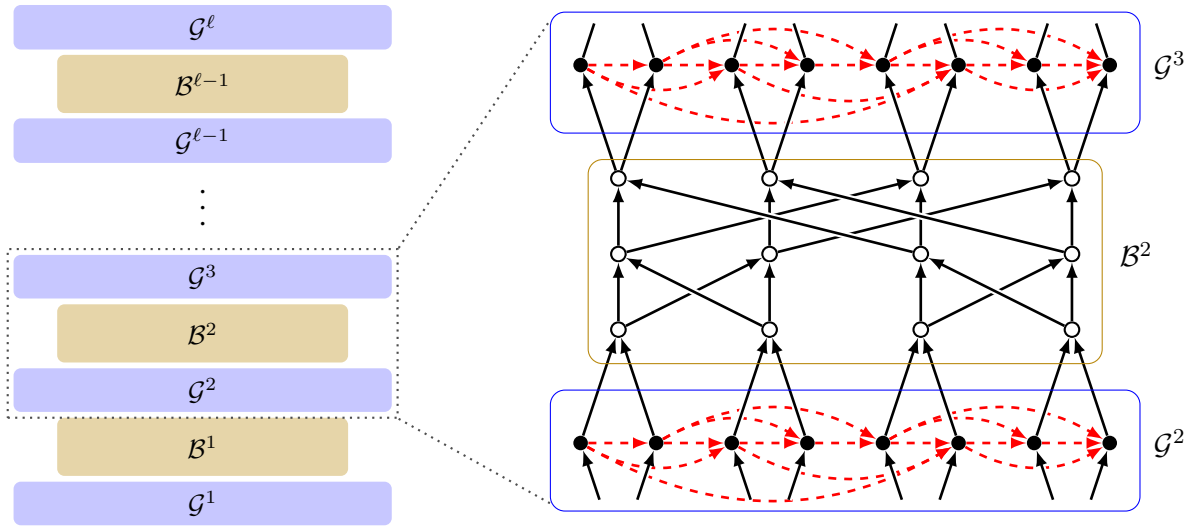
By connecting multiple copies of a DRG with butterfly networks, we can achieve the compression-robustness necessary to build a PIE. Specifically, if $\mathcal{G}$ is a (γ, δ)-DRG, we need $\ell = \lceil 1/\gamma \rceil + 1$ layers of $\mathcal{G}$ to produce a $(1, \delta)$-compression-robust graph. We refer to this multi-layer construction, shown graphically in Figure 4, as a *Dagwood Sandwich Graph* (DSaG). We define a DSaG formally in Construction 2.

As mentioned above, this DSaG is a $(1, \delta)$-compression-robust encoder graph.

Theorem 2. *For any (γ, δ)-depth-robust graph $\mathcal{G}$ with $n = 2^k$ nodes, the associated DSaG $\widehat{\mathcal{G}}$ is $(1, \delta)$-compression-robust.*

Proof Sketch. Let $\widehat{\mathcal{G}}'$ denote $\widehat{\mathcal{G}}$ after removing data edges $\tilde{E}_D$ and their corresponding key edges.

For each copy $\mathcal{G}^i$ of the DRG, we model a vertex as being removed unless it retains a path to a non-removed vertex in $\mathcal{G}^{i+1}$ (or an output vertex for $\mathcal{G}^\ell$). By induction on i from $i = \ell$ down to $i = 1$, this guarantees that any non-removed vertex retains a path to some sink node of $\widehat{\mathcal{G}}$. We then bound the number of vertices removed from each $\mathcal{G}^i$ based on the number of vertices removed from $\mathcal{G}^{i+1}$ and the number of



(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 butterfly graphs ($\mathcal{B}^i$), while the right shows the structure of the edges, with data edges in solid black and key edges in dashed red.

edges in $\tilde{E}_D$ between the two. Using this bound, we show that if $|\tilde{E}_D| < n$ and $\ell \geq \frac{1}{\gamma} + 1$, there is some i for which fewer than γn vertices are removed from $\mathcal{G}^i$.

Since $\mathcal{G}^i$ is (γ, δ)-depth-robust, the non-removed portion of $\mathcal{G}^i$ must contain a path P_K of length at least δn. Each edge in P_K is a key edge in $\widehat{\mathcal{G}}'$, and the terminal vertex of P_K has a path P' to an output vertex (or it would have been removed). The concatenation $P_K \parallel P'$ forms exactly the path needed to prove compression-robustness. □

We provide a complete proof of Theorem 2 in Appendix C.

Taken together with Theorem 1, this means a DSaG is sufficient to construct a PIE.

6 Optimizations

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

6.1 Removing Unnecessary Key Derivations

The proof that a compression robust graph is incompressible 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

Construction 2 (Dagwood Sandwich Graph)**.** Let $n = 2^k$, $\mathcal{G} = (V, E)$ be a (γ, δ)-DRG, and $\mathcal{B} = \mathcal{B}_{k-1}$ be a butterfly network on $n/2$ elements. Let $\ell = \lceil 1/\gamma \rceil + 1$. We construct a *Dagwood sandwich graph* $\widehat{\mathcal{G}}$ as follows.

Let I and O be sets of n input and output vertices, respectively. Let $\mathcal{G}^1, \ldots, \mathcal{G}^\ell$ be ℓ distinct copies of $\mathcal{G}$, and let $\mathcal{B}^1, \ldots, \mathcal{B}^{\ell-1}$ be $\ell - 1$ distinct copies of $\mathcal{B}$. The vertices of $\widehat{\mathcal{G}}$ are I, O, and the vertices of each $\mathcal{G}^i$ and each $\mathcal{B}^i$.

The edges of $\widehat{\mathcal{G}}$ consist of all edges in each $\mathcal{G}^i$ and each $\mathcal{B}^i$ as well as the following:

- For $j \in [1, n]$, add an edge from the jth vertex of I to the jth vertex of $\mathcal{G}^1$ and an edge from the jth vertex of $\mathcal{G}^\ell$ to the jth vertex of O.
- For $j \in [1, n/2]$ and $i < \ell$, add edges from vertices $(2j - 1)$ and $2j$ of $\mathcal{G}^i$ to the jth input vertex of $\mathcal{B}^i$, and edges from the jth output vertex of $\mathcal{B}^i$ to vertices $(2j - 1)$ and $2j$ of $\mathcal{G}^{i+1}$.

The key edges $\widehat{E}_K$ are the edges originally contained in the DRGs $\mathcal{G}^1, \ldots, \mathcal{G}^\ell$, and the data edges $\widehat{E}_D$ are all others.

derivation function KDF′ at these vertices. We still require that KDF′ act 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 SHA256, while we still need a memory-hard 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 5 contains *mostly* vertices with no incoming key edges, this optimization makes a dramatic impact.

6.2 Parallelizing Decoding

While the ENCODE operation of a PIE must inherently be slow, the DECODE operation need not. In fact, as files must be decoded before they are used, it is a significantly advantage to support fast decoding. Our PIE construction supports rapid decoding through parallelism.

We enforce inherently sequential work in ENCODE by using the output of one computation in a DRG layer of a DSaG as the input to KDF for other computations in the same layer. Thus, in order to perform the second computation, the first must be complete. The data values output by a vertex during encoding are, however, the values *input* to that vertex during decoding. Thus, 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 a butterfly graph, they can easily be made available in parallel. Therefore, while decoding, we can perform every KDF call on a given DRG layer at the same time, massively reducing the time necessary to decode the file.

As we show in Section 8.3, in practice this can result in an order of magnitude speedup of DECODE.

6.3 Parallelizing Encoding

Unlike decoding, 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.

To bypass this impediment and allow for parallelism, we note that, for any $(1, \delta)$-compression-robust n-encoder graph $\mathcal{G}$, m disconnected copies of $\mathcal{G}$ together form a $(1, \delta/m)$-compression-robust mn-encoder graph. Removing fewer than n data edges (and their corresponding key edges) from any one copy of $\mathcal{G}$ must

leave a path with at least δn key edges in that copy. Therefore it requires mn data edges to remove all such paths from all m copies of $\mathcal{G}$. As the number of key edges in the long path (δn) does not change but the graph size does, we are left with $(1, \delta/m)$-compression-robustness. Importantly, because the copies of $\mathcal{G}$ are disconnected, we can split the file into n-block chunks and encode those chunks in parallel.

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

Regardless of the parameterization, any fixed chunk size n allows the encoding time to scale efficiently with large file size. Specifically, the cost of encoding any F is now $O\left(\frac{|F|}{n} \cdot |\mathcal{G}|\right)$—linear in $|F|$. Using our DSaG construction, $|\mathcal{G}| = O(n \log n)$, meaning $\text{ENCODE}_{\mathcal{G}}(F)$ runs in time $O(|F| \log n)$.

7 Instantiation

In order to instantiate a PIE, we apply Construction 1 to the DSaG defined by Construction 2. Constructing the DSaG, however, requires a depth-robust graph (DRG) with known constants.

7.1 Security Efficiency Ratio

Before choosing a DRG, we need to know not just the properties of the DRG itself, but also how those properties will affect the efficiency of our PIE. One way to measure the efficiency of a PIE is by comparing execution time of ENCODE with the guaranteed sequential work needed to recompute a discarded block. We call this ratio the *security efficiency ratio* (SER). An SER of 1 would indicate that the time needed for ENCODE is the same as the time needed to recompute a single discarded block—a bound no secure PIE can ever break. The SER therefore measures how far ENCODE's performance is from the theoretical optimum.

If a PIE allows parallelism during encoding, its SER will vary with parallelism. For our construction, DSaGs preclude parallelism within a file chunk, but the cutting files into fixed-size independent chunks (Section 6.3) allows for complete parallelism across chunks. We thus attempt to minimize the sequential SER of a single chunk.

For a PIE built using any compression-robust graph $\mathcal{G}$ and Construction 1, we can bound the (sequential) SER by examining the graph. Ideally the cost of $\text{ENCODE}_{\mathcal{G}}$ will be dominated by the number of calls to KDF. Using the fast KDF′ optimization from Section 6.1, we need one KDF operation for each vertex with an incoming key edge. For a $(1, \delta)$-compression-robust graph $\mathcal{G}$ with vertices V and key edges E_K, the SER of $\text{ENCODE}_{\mathcal{G}}$ is therefore at least $\frac{1}{\delta n}|\{v \in V \mid \exists (u, v) \in E_K\}|$.

In a DSaG, nearly every vertex in a depth-robustness layer has incoming key edges, while no vertices in butterfly layers do. This means that a DSaG with ℓ layers will have SER at least $\ell n/\delta n = \ell/\delta$. As per Construction 2, ℓ and δ, and thus the SER bound, are determined entirely by the underlying DRG. Specifically, if that graph is (γ, δ)-depth-robust, then $\ell = \lceil 1/\gamma \rceil + 1$, meaning the SER of the DSaG is at least

$$\frac{\frac{1}{\gamma} + 1}{\delta} = \frac{\gamma + 1}{\gamma\delta}.$$

Notably, if γ and δ are both constants—which is often the goal of DRG constructions—this ratio is constant in n.

We can use this to determine what DRG constants will produce the most efficient PIEs. No graph can have depth-robustness better than $(\gamma, 1-\gamma)$, so we must find γ to minimize $\frac{\gamma+1}{\gamma(1-\gamma)}$. As we always ensure $\ell \leq \frac{1}{\gamma} + 1$, we can instead minimize $\frac{\ell(\ell-1)}{\ell-1}$. Since ℓ must be an integer and $\ell \geq 3$, this is minimal at $\ell = 3$ or 4, either of which result in a lower bound of 6 on the SER.

This bound on the SER may be loose for two reasons: overhead from fast operations (KDF′ and PRP) 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 in $\mathcal{G}$. The in-degree of a vertex in $\mathcal{G}$ determines the input length of the corresponding KDF call. Longer inputs may take more time, but since we cannot force recomputation of any specific values, our wall-clock security bound must conservatively assume the minimum input length for KDF. As this bound is definitionally the "security" portion of SER, larger in-degree graphs may have higher SER for a given γ and δ.

7.2 DRG Construction

A variety of prior work has explored how to efficiently construct DRGs [1, 31]. These constructions aim to maximize γ and δ while minimizing in-degree and complexity of constructing the graph initially. Despite good asymptotic bounds, the graphs are not practical in our construction.

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

Another option is a structure proposed by Mahmoody et al. [31] that deterministically produces $(\gamma, 1 - \gamma - \varepsilon)$-DRGs with in-degree $\widetilde{O}(\log^2 n)$ for any $\gamma \in (0, 1)$ and any $\varepsilon > 0$. While such graphs could result in very small SER in theory, Mahmoody et al. do not provide sufficient detail to compute the precise in-degree or implement the construction.

We instead rely on a naïvely-constructed DRG. Specifically, for any $\gamma \in (0, 1)$, we can build a $(\gamma, 1-\gamma)$-DRG with n vertices and in-degree γn as follows. For each $i \in [1, n]$, the parents of v_i are $v_{i-\gamma n}, \ldots, v_{i-1}$. This guarantees that, for any set of k consecutive vertices $v_i, \ldots, v_{i+k-1}$ with $k < \gamma n$, vertex v_{i-1} connects to v_{i+k} since $i - 1 + \gamma n > i - 1 + k$. Therefore removing fewer than γn vertices leaves a path with all $(1 - \gamma)n$ remaining vertices. This graph's in-degree scales poorly with n, but if we encode large files in multiple independent chunks—as described in Section 6.3—there is no need for large DRGs. Moreover, we see in Section 8 that we can tune KDF to provide very practical timing guarantees on graphs small enough that ENCODE's runtime is dominated by KDF's memory-hardness, not the graph in-degree.

Using the values computed above to set the SER to 6 while also minimizing in-degree, we set $\gamma = 1/3$, giving $\ell = 4$.

8 Experiments

We now present performance results for our implementation of the PIE using the DRG specified in Section 7.2.

Our PIE implementation is only 325 lines of Java code that rely on BouncyCastle [9] for all hash and cipher primitives. We measured performance for 128 and 256-bit blocks, requiring 128, 256, and 512-bit PRPs. We use AES as a 128-bit PRP, and Threefish [23] with appropriate block size for the other two.[4] We

[4] We chose Threefish because BouncyCastle provides a fast implementation that supports both 256-bit blocks and 512-bit blocks.

Block Size	1 Block PRP	2 Block PRP	KDF′
128-bit	0.65μs	0.19μs	0.43μs
256-bit	0.19μs	0.31μs	0.58μs

Table 1: The runtime for a single iteration of each fast operation.

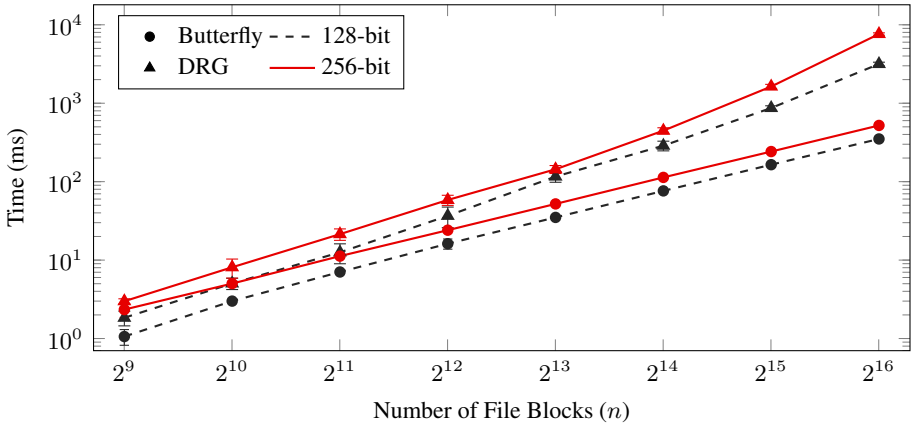


Figure 5: Performance of butterfly network transforms and DRG transforms for multiple graph and block sizes.

instantiate KDF as the memory-hard hash function scrypt [33], and use SHA256 or SHA512 (depending on block size) as the fast KDF′ in the optimization in Section 6.1. We ran all experiments on a single `c5.18xlarge` Amazon EC2 instance, which includes two Intel Xeon Platinum 8124M CPUs, each of which contain 18 hyperthreaded cores.

8.1 Micro-benchmarks

In our security model, we model all operations except KDF as instantaneous. To confirm that they add minimal overhead in practice, we benchmarked each of these operations separately for both 128-bit blocks and 256-bit blocks. The results in Table 1 show that these operations are extremely fast and are unlikely to impose meaningful overhead.

We further benchmarked single iterations of both the butterfly network and DRG transforms within our PIE. For each transform, we measured the runtime for a variety of graph sizes with both 128 and 256-bit blocks. As the DRG layer usually invokes the slow KDF, we replace scrypt with a fast random number generator for this benchmark. Even though the butterfly structure lends itself to parallelism, the operations are efficient enough that we perform them sequentially. The results in Figure 5 show that each layer imposes small overhead for reasonably-sized graphs. The poor scaling of the DRG layer is due to the linear in-degree of our naïve graph, resulting in large memory copies. As we confirm below, this overhead is minimal for small graphs and expensive KDF's.

Chunk Size	KDF Mem	Min Seq Work	ENCODE Time	SER
32 KiB	256 KiB	0.59 sec	4.0 sec	6.9
	512 KiB	1.1 sec	7.4 sec	6.5
	1 MiB	2.3 sec	14.3 sec	6.2
64 KiB	256 KiB	1.2 sec	9.1 sec	7.7
	512 KiB	2.3 sec	15.8 sec	6.8
	1 MiB	4.6 sec	29.5 sec	6.5
128 KiB	256 KiB	2.4 sec	22.2 sec	9.3
	512 KiB	4.6 sec	35.6 sec	7.7
	1 MiB	9.2 sec	63.1 sec	6.9

Table 2: Performance of ENCODE for our PIE on a single file chunk of various sizes using 256-bit blocks and various KDF difficulties.

8.2 Basic PIE Performance

In order to ensure that the formal security of a PIE corresponds to practical security, we need KDF to be sufficiently slow. In particular, if an encoding is d-oracle incompressible, the time required to execute KDF d times in sequence must be noticeable. By using different parameters for KDF—in this case scrypt—we can tune this value.

Table 2 presents performance measurements for encoding a single file chunk of various sizes with multiple scrypt parameterizations.[5] The minimum sequential work was measured by timing multiple sequential executions of scrypt on minimum-length inputs. The SER was then computed by dividing the total encoding time by this minimum sequential work. Thus any slowdown caused by high in-degree of the graph or slow PRP operations is reflected as a higher SER.

Parameterizing the minimum sequential work to outstrip any reasonable network latency, we attain nearly optimal SER on small file chunks. Given that ENCODE must inherently be a somewhat slow operation—and should thus be uncommon—we believe this performance is very practical.

8.3 Parallel Operations

As we described in Section 6, we can parallelize our PIE construction across independently-encoded chunks of the file for ENCODE, and, more aggressively, across KDF calls within a file chunk for DECODE.

Parallelizing ENCODE. As discussed in Section 6.3, we can split a file into independent chunks and encode each chunk separately. For this optimization to work, the file must be large enough that each chunk provides a sufficient sequential work guarantee by itself. We measured the throughput of a parallel implementation of ENCODE for varying levels of parallelism. We configured ENCODE to use 256-bit blocks, 32 KiB file chunks, and 1 MiB of memory in scrypt, ensuring 2.3 seconds of inherently sequential work. This parameterization allowed for effective parallelism on files of only hundreds of kilobytes.

[5] Scrypt takes three parameters: work factor, block multiplier, and independent instances. For every invocation we set the block multiplier to 8 (resulting in a 1 KiB block size) and use 1 independent instance. We vary only the work factor. Note that scrypt's total memory use is (work factor) $\times$ (block size).

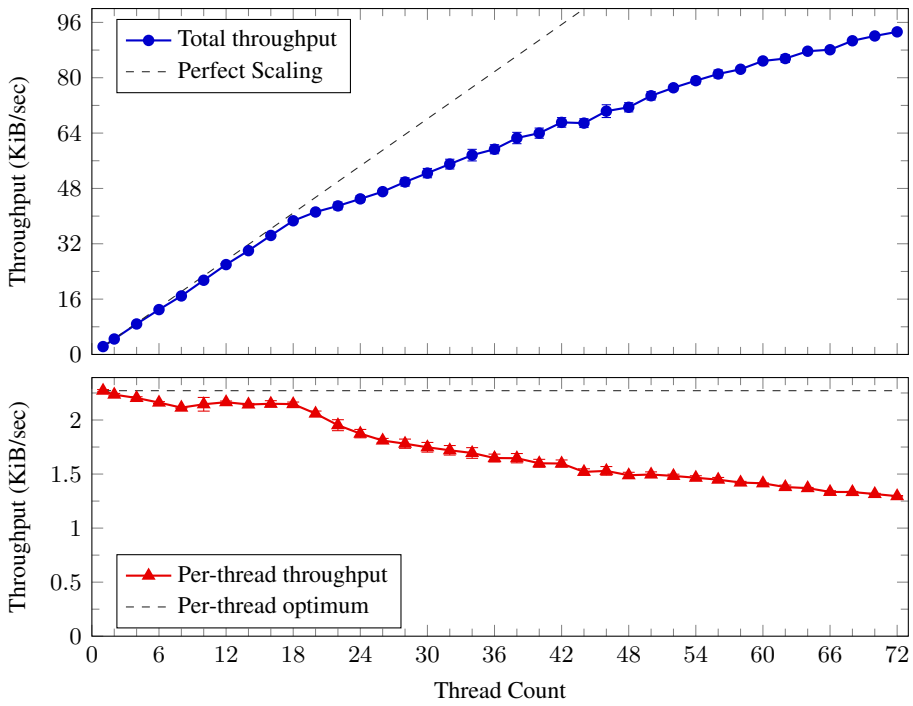


Figure 6: Parallel throughput of ENCODE with various thread counts. The top plot shows the total system throughput as a function of the number of threads, while the bottom shows the average per-thread throughput. All tests were run on a `c5.18xlarge` Amazon EC2 instance with each thread encoding an independent 32 KiB chunk of 256-bit blocks with scrypt tuned to use 1 MiB (resulting in a 2.3 second minimum on inherently sequential work).

Figure 6 shows that parallelism is very effective. While the benefits of more threads appear to decrease around 18-20 threads, we believe this is due to the `c5.18xlarge`'s hardware configuration. With two CPUs with 18 hypthreaded cores, at 18-20 threads it must either place multiple ENCODE operations on the same physical core or spread them across multiple CPUs which must maintain memory consistency, either of which will reduce throughput. Each CPU also has 24.75 MiB of L3 cache, so with scrypt tuned to 1 MiB, 18-20 threads is likely around the point when the memory usage exceeds the L3 cache of a single CPU, requiring some operations to hit main memory. As each chunk can be encoded completely independently, we could avoid such hardware limitations by employing multiple machines.

Parallelizing DECODE. Section 6.2 explains how our DSaG construction enables DECODE to operate in parallel, even for small single-chunk files. To measure this benefit, we timed the performance of parallel decoding for a single 32 KiB file chunk. We again used 256-bit blocks and tuned scrypt to use 1 MiB of memory. The results in Figure 7 demonstrate that parallel decoding is very efficient. With enough threads, decoding a full chunk took less than half of the 2.3 second minimum required to re-encode any discarded blocks.

The runtime for DECODE stopped improving around 18 threads, which we believe has two causes. First, as with parallel encoding, the hardware configuration of the `c5.18xlarge` instance reduces parallel efficiency beyond 18 threads. Second, parallelism within a single chunk is inherently limited by the chunk size. For files with multiple chunks, we can decode each chunk independently as with ENCODE.

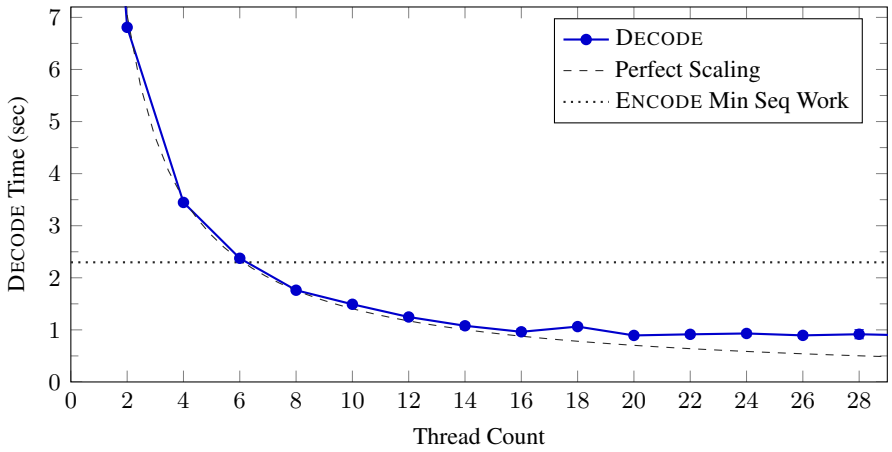


Figure 7: Running time of DECODE for a 32 KiB chunk with various numbers of threads on a `c5.18xlarge` Amazon EC2 instance. Scrypt is tuned to use 1 MiB of memory. Note that decoding one chunk with 8 threads is faster than the inherently sequential work to needed re-encode any discarded data.

9 Related Work

Proofs of Storage. As noted above, PoRs [19, 29, 38] and PDPs [4], often generically called proofs of storage [30], prove only file retention by a provider, not file replication. As PoRs involve erasure coding, they may be viewed as proofs of replication in the trusted-encoder, private-reader setting. Multi-replica PDPs, e.g., [15], may be viewed similarly.

Proofs of space, introduced by Dziembowski et al. [21] as an alternative to proof of work in blockchains, are conceptually related. They help ensure that a prover is consuming a certain amount of storage. Proofs of space, however, involve specially constructed files, not generic files as in a DSN.

Proofs of Replication. Several early-stage DSNs, such as Sia [42] and Storj [43], already perform proofs of replication in the trusted-encoder, private-reader setting.

An encoding technique in the trusted-encoder, public-reader setting using the RSA trapdoor one-way permutation was proposed in van Dijk et al. [41]; that construction was not intended for replication, but could be used to prove it. Damgård, Ganesh, and Orlandi [17] propose a similar construction, but embellish it using PoRs to support file extraction, rather than just incompressibility, and offer a more in-depth analysis focused on file replication.

As discussed above, there have been previous attempts to construct PIEs [6, 34, 41]. In addition to these works, in a recent presentation, Fisch et al. [24] first proposed use of DRGs to address the untrusted-encoder, public-reader file replication problem, using a security notion different from a PIE. To date they have not published details. Nonetheless, our use of DRGs was inspired by their insight.

Depth-Robust Graphs (DRGs). DRGs were studied decades ago for proving lower bounds on computational complexity [22, 36, 37]. DRGs have since proven useful for enforcing sequential work in publicly verifiable proofs of work [31] and memory-hard hash functions [1–3, 14], prompting a revival of interest as well as new DRG constructions [1, 31].

Incompressible Functions. Various notions of incompressibility used to prove the security of cryptographic constructions, e.g., [28], and protect keys for digital-rights management [20], do not relate directly to our

setting. More relevant are storage-enforcing commitments [27], which bound compressibility to prove that a certain, minimal amount of storage is devoted to a file F. They do not enable proof of replication, though.

Publicly Verifiable Proofs of Sequential Work and Delay Functions. Publicly verifiable proofs of sequential work, including those introduced by Cohen and Pietrzak [13] and Mahmoody et al. [31], which uses DRGs, are public-coin systems that enable a prover to show that it has done a certain amount of sequential work relating to a particular input. They enable fast verification (polylog in the time of the prover), which our construction does not. Intended for applications that do not involve data recovery (e.g., timestamping, CPU benchmarks), they do not enforce uniquely computable outputs and do not support decoding, as needed for proofs of replication.

Delay functions, introduced by Goldschlag and Stubblebine [26], are similar, but do enforce uniquely computable outputs and can be decodable. Boneh et al. [8] show how to construct verifiable delay functions (VDFs) that are decodable and, in principle, useable for proofs of replication. They show how to make verification fast (again, polylog in the prover's time) by relying on a SNARK framework called incremental verifiable computation (IVC)—something not possible in our DSaG construction. Their constructions are theoretical, however, and they report no implementation. Among other issues, the proposed VDFs involve chained operations on individual (e.g., 256-bit) field elements composing a file, so their SERs (see Section 7.1) are impractically high (e.g., SER $\approx$ 8,000 for a 128 KiB file).

10 Conclusion

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

A number of interesting open problems arise from our work. There is the challenge of enabling efficient verification of correct PIE computation without impractically heavyweight proof systems as used by Boneh et al. [8]. There is also the problem of supporting *dynamic* provable replication: replicating files whose contents may change over time. This is trivially achievable by updating modified file chunks, but could be impractical for sparsely distributed file changes such as in a database. File-system approaches such as log-structured updates may prove helpful, but encoder-level support could provide interesting, complementary mechanisms. Finally, developing more efficient DRGs would allow for more efficient large PIEs in theory, but this is unlikely to significantly improve practical performance.

In summary, we believe that the PIE constructions we present here are eminently practical for a range of secure storage applications: file replication in DSNs, redundant file distribution in permissioned blockchains, proof of at-rest file encryption, and much more.

Acknowledgements

Many people helped with this work. Nicola Greco and Juan Benet at Protocol Labs helped structure the problem and clarify practical requirements. Ben Fisch and Joe Bonneau alerted us to the utility of depth-robust graphs, and their concurrent work pushed us toward stronger adversarial models. Mic Bowen at Intel and Samarth Kulshreshtha pointed us to new potential applications of PIEs. Yuwen Wang provided insight

that led to the proof of Lemma 1. Finally, Lorenz Breidenbach, Rahul Chatterjee, and Paul Grubbs asked many insightful questions and helped with editing.

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

References

[1] J. Alwen, J. Blocki, and B. Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In *CCS*. ACM, 2017.

[2] J. Alwen, J. Blocki, and K. Pietrzak. Depth-robust graphs and their cumulative memory complexity. In *EUROCRYPT*. Springer, 2017.

[3] 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 *AsiaCCS*, pages 51–65. ACM, 2018.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *CCS*. ACM, 2007.

[5] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 863–874, 2013.

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

[7] 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.

[8] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. *IACR Cryptology ePrint Archive*, June 2018. `https://eprint.iacr.org/2018/601`.

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

[10] 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 *CCS*. ACM, 2011.

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

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

[13] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In *EUROCRYPT*. Springer, 2018.

[14] H. Corrigan-Gibbs, D. Boneh, and S. E. Schechter. Balloon hashing: Provably space-hard hash functions with data-independent access patterns. *IACR Cryptology ePrint Archive*, 2016. https://eprint.iacr.org/2016/027.

[15] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *ICDCS*. IEEE, 2008.

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

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

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

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

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

[21] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In *CRYPTO*. Springer, 2015.

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

[23] 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.

[24] 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.

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

[26] D. M. Goldschlag and S. G. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In *Financial Cryptography and Data Security*. Springer, 1998.

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

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

[29] A. Juels and B. S. Kaliski Jr. PORs: Proofs of retrievability for large files. In *CCS*. ACM, 2007.

[30] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security*. Springer, 2010.

[31] M. Mahmoody, T. Moran, and S. Vadhan. Publicly verifiable proofs of sequential work. In *ITCS*. ACM, 2013.

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

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

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

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

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

[37] G. Schnitger. On depth-reduction and grates. In *FOCS*. IEEE, 1983.

[38] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*. Springer, 2008.

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

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

[41] 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 *CCS*. ACM, 2012.

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

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

Appendix

A Multi-Query Public Incompressible Encodings

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 number of files $m \geq 1$, any compression factor ϵ, any number of queries s, and any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function $negl$ such that

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

where

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

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

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

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

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

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

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

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

endfor

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

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

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

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

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

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

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

Since s is polynomial in λ, there are negligible functions $negl^\star$, $negl''$ such that

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

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

B Compression-Robustness Gives Incompressibility

We now prove that a $(1, \delta)$-compression-robust n-encoder graph produces a δn-oracle incompressible.

Proposition 2. *Let $\mathcal{G} = (V, E)$ be an (γ, δ)-compression-robust n-encoder graph with data edges E_D. Let $\tilde{E} \subseteq E_D$ and $\mathcal{G}'$ be the graph induced by removing $\tilde{E}$ and all corresponding key edges as defined in Definition 3. For all ϵ, if $\frac{|\tilde{E}|}{n} \leq \gamma - \epsilon$, then there exists some set of sink nodes V_s such that $|V_s| \geq \epsilon n$ and for all $v \in V_s$ there is a path in $\mathcal{G}'$ ending in v that contains at least δn key edges.*

Proof. Without loss of generality, assume $\epsilon > 0$. If $\epsilon \leq 0$, the statement is trivial by letting $V_s = \emptyset$. We also note that it suffices to prove that, for all $\tilde{E}$, there exist ϵn sink nodes in $\mathcal{G}$ terminating paths containing at least δn key edges. The rest of the proof will follow by induction on $\lceil \epsilon n \rceil$.

If $\lceil \epsilon n \rceil = 1$, the proposition follows directly from the definition of (γ, δ)-compression-robustness.

We take as our inductive hypothesis that $\lceil \epsilon n \rceil \geq 2$ and for all $\tilde{E}' \subseteq E_D$ with $|\tilde{E}'| \leq \gamma n - (\epsilon n - 1)$, there are at least $\epsilon n - 1$ sink nodes terminating paths with δn key edges. Let $\tilde{E} \subseteq E_D$ and let V_s be all sink nodes in $\mathcal{G}$ terminating paths in $\mathcal{G}'$ with δn key edges. Assume for the sake of contradiction that $|\tilde{E}| \leq n(\gamma - \epsilon)$ but $|V_s| < \epsilon n$. Let $v \in V_s$ be any vertex vertex terminating a relevant path, and let $(u, v) \in E_D$ be the sole edge coming into v. Let $\tilde{E}' = \tilde{E} \cup \{(u, v)\}$. We now claim that $\tilde{E}'$ violates our inductive hypothesis.

Since (u, v) is the only edge into v and $v \in V_s$, it must be the case that $(u, v) \notin \tilde{E}$. Therefore

$$|\tilde{E}'| = |\tilde{E}| + 1 \leq n(\gamma - \epsilon) + 1 = \gamma n - (\epsilon n - 1).$$

Now let V'_s be the set of sink nodes in $\mathcal{G}$ terminating paths containing d key edges in $\mathcal{G}''$, the graph induced by removing $\tilde{E}'$ and all corresponding key edges. Since $\tilde{E} \subset \tilde{E}'$ and removing more edges cannot add more paths, $V'_s \subseteq V_s$. Moreover, $(u, v) \in \tilde{E}'$ so there are no nontrivial paths to v in $\mathcal{G}''$, and thus $v \notin V'_s$. It follows that $V'_s \subseteq V_s \setminus \{v\}$, so

$$|V'_s| \leq |V_s \setminus \{v\}| = |V_s| - 1 < \epsilon n - 1.$$

This contradicts our inductive hypothesis that $|V'_s| \geq \epsilon n - 1$. Hence our assumption that $\frac{|\tilde{E}|}{n} \leq \gamma - \epsilon$ but $|V_s| < \epsilon n$ must be false, thus proving the inductive case and the proposition. □

We now use this in our proof of Theorem 1.

Theorem 1. *If $\mathcal{G}$ is an $(1, \delta)$-compression-robust n-encoder graph, then $\text{ENCODE}_{\mathcal{G}}$ as defined in Construction 1 is a δn-oracle public incompressible encoding.*

Proof. Let $\mathcal{G} = (V, E)$ be a $(1, \delta)$-compression-robust graph with data edge E_D and key edge E_K. We model KDF^i as a family of random oracles and PRP^i_κ as ideal ciphers.

Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be defined as in Definition 1. When we refer to "$\mathcal{A}$ storing data" we mean that it is explicitly included in G' as output by $\mathcal{A}_1$.

We first claim that $\mathcal{A}$ can gain no advantage by storing data not represented by a data edge in $\mathcal{G}$. By assumption, $\mathcal{A}$ can learn nothing of the outputs of KDF or PRP without computing them, and the only operations performed by $\text{ENCODE}_{\mathcal{G}}$ concatenating or splitting bit strings, KDF, and PRP. It is therefore the

case that $\mathcal{A}$ cannot possibly gain any advantage by storing data that is not an input to KDF or PRP (including keys output by KDF and used by PRP).

This leaves three potentially relevant types of values: (*i*) global keys and randomness, (*ii*) derived PRP keys, and (*iii*) data explicitly represented by edges in $\mathcal{G}$. There is no reason to store global keys or randomness, as they are provided to $\mathcal{A}_2$ explicitly. There is no advantage to storing derived key data. Each key is of length at least λ and is derived by calling a random oracle in inputs that are guaranteed to be different. Within a file, the tweak parameter changes at each vertex. Across files, the file-level encoding key $\kappa_i = \mathcal{O}(\rho_i \parallel F_i)$, and thus $\mathcal{A}$ can find a collision with only negligible probability. Therefore the probability of any two vertex keys being equal is negligible. Moreover, keys are always the same length as the output of the PRP for which they are used. As $\mathcal{A}$ cannot compress the derived keys themselves and those keys are the same length as the values they are used to permute, $\mathcal{A}$ can simply store the values instead. The only data $\mathcal{A}$ can gain an advantage by storing, therefore, is data explicitly represented by edges in $\mathcal{G}$.

Consider the case where $\mathcal{A}$ stores only whole blocks of edge data. We demonstrate later that it can do no better by storing partial blocks. The value of each key edge (u, v) in Construction 1 is identical to the value of the outgoing data edge from u. We can thus represent any full-block data storage by $\mathcal{A}$ as removal of data edges in $\mathcal{G}$ along with any corresponding key edges, as defined in Definition 3. As PRP_κ is an ideal cipher and no two vertices use the same key, the only way for two outputs to be the same (with non-negligible probability) is if at least one input is computed by inverting PRP_κ on the shared output. Propagating this back to the inputs, $\mathcal{A}$ would need the file-level key κ_i to compute the value of the file F_i. As $\kappa_i = \mathcal{O}(\rho_i \parallel F_i)$, this can only happen with negligible probability. Thus, across all m files, no two non-input data edges represent the same data except with negligible probability.

This means that, if $\mathcal{A}$ stores k blocks of data, that data must correspond to at most k distinct data edges. Letting G, G' as defined in $\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}(\lambda, m, \epsilon, d)$, G' must therefore corresponds to some set of data edges $\tilde{E}_D \subset E_D$ where $|\tilde{E}_D| \leq |G'|$. Hence

$$\frac{|\tilde{E}_D|}{n} \leq \frac{|G'|}{|G|} \leq 1 - \epsilon.$$

Let $\mathcal{G}' = (V, E')$ be the subgraph if $\mathcal{G}$ attained by removing $\tilde{E}_D$ and its corresponding key edges. By the compression-robustness of $\mathcal{G}$ and Proposition 2, there are at least ϵn output vertices v in $\mathcal{G}'$ that terminate paths $P_v \subseteq E'$ containing at least δn key edges. We now claim that $\mathcal{A}_2^{\mathcal{O}^\star_{\delta n}}$ has a negligible probability of recomputing any of these outputs.

By the ideal cipher property of PRP, in order for $\mathcal{A}_2$ to determine the value output along an edge, it must either have the value stored or recompute it. To recompute the value, it needs all inputs (both data and the key). Similarly, since KDF is modeled by $\mathcal{O}$, $\mathcal{A}_2$ must invoke $\mathcal{O}^\star_{\delta n}$ with all key inputs to derive the key for any vertex it needs to recompute. Thus a path in $\mathcal{G}'$ represents a sequence of values $\mathcal{A}_2$ must recompute where the input to each computation requires the output of the previous. In particular, if the path contains at least δn key edges, $\mathcal{A}_2$ cannot invoke $\mathcal{O}^\star_{\delta n}$ enough times to use it to recompute every value in the path.

Let v_j be an output vertex of $\mathcal{G}'$ terminating a path $P_{v_j} \subseteq E'$ with at least δn key edges. As argued above, $\mathcal{A}_2$ cannot compute the output of this path by calling $\mathcal{O}^\star_{\delta n}$ to derive every necessary key. The correct keys, however, are all at least λ bits long and are derived by $\mathcal{O}$, which is a random oracle. Therefore, for at least one key along the path $\mathcal{A}_2$ can do no better than a random guess for that key, which will be correct with probability $2^{-\lambda}$.

While this probability is negligible, we note that each step along the path presents an opportunity for a collision. If another call to $\mathcal{O}$ or PRP outputs a value that collides with the original correct value $\mathcal{A}_2$ will still succeed. Each such call has a collision probability of (at most) $2^{-\lambda}$. At each vertex in the path there is one call

to $\mathcal{O}$ and one call to PRP, giving two chances at a collision. The length of the path, however, is bounded by the number of vertices V, which is polynomial in λ. Thus using the union bound ($\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$), we see that for some polynomial $poly(\lambda)$,

$$\Pr\left[\mathcal{A}_2^{\mathcal{O}^\star_{\delta n}}(j, G', \{(\rho_i, \kappa_i)\}_{i=1}^m = G[j]\right] \leq poly(\lambda) \cdot 2^{-\lambda}.$$

Therefore, if $\mathcal{A}$ stores only complete blocks of data, it can do no better than

$$\begin{aligned}\Pr\left[\text{EXP}^{\text{PIE}}_{\mathcal{A},\mathcal{O}}(\lambda, m, \epsilon, d) = 1\right] &\leq (1-\epsilon) + \frac{poly(\lambda)}{\epsilon n} \cdot 2^{-\lambda} \\ &= (1-\epsilon) + negl(\lambda).\end{aligned}$$

It is now left to show that $\mathcal{A}$ can do no better by storing partial blocks. If $\mathcal{A}$ discards a single bit from each of k blocks with $k \geq 2$, it has a lower probability of success than if it discarded the entirety of a single block. By discarding a full single block $\mathcal{A}_2$ can guess on that block giving it a success probability of at least $\frac{n-1}{n} + 2^{-\lambda}$. By discarding bits from k blocks $\mathcal{A}$ is forced to "recompute" values to at least k output vertices, which it may be able to do by guessing a single discarded bit and recomputing the rest of a path (that may contain fewer than δn key edges). The probability of correctly "recomputing" such a node is at most $\frac{1}{2}$ plus the collision probability $poly(\lambda) \cdot 2^{-\lambda}$. Thus

$$\begin{aligned}\Pr[\mathcal{A}_2 \text{ succeeds}] &\leq \frac{n-k}{n} + \frac{k}{2n} \cdot poly(\lambda) \cdot 2^{-\lambda} \\ &= 1 - \frac{k}{2n}(2 - poly(\lambda) \cdot 2^{-\lambda}) \\ &\leq 1 - \frac{1}{n}(2 - poly(\lambda) \cdot 2^{-\lambda}) \\ &= 1 - \frac{1}{n} + \frac{1}{n}(poly(\lambda) \cdot 2^{-\lambda} - 1).\end{aligned}$$

Since $poly(\lambda) \cdot 2^{-\lambda} \ll 1$, this simplifies to $\Pr[\mathcal{A}_2 \text{ succeeds}] < 1 - \frac{1}{n}$, which is less than the success probability when discarding one entire block. As discarding a single bit from k blocks is always worse than discarding all of $\frac{k}{2}$ blocks, $\mathcal{A}$ cannot gain an advantage by discarding partial blocks. □

C DSaGs are Compression-Robust

We now present a proof that the DSaG described in Construction 2 is compression-robust. To prove Theorem 2 we rely on the following two lemmas bounding the extent to which an adversary can disconnect the graph by removing edges inside butterfly networks.

Lemma 1. *Let $\mathcal{B}_k = (V, E)$ be a butterfly network with $n = 2^k$ input nodes $I = \{v \in V \mid \text{DEG}_{\text{in}}(v) = 0\}$ and n output nodes $O = \{v \in V \mid \text{DEG}_{\text{out}}(v) = 0\}$.*

Given $\tilde{V}_O \subseteq O$ and $\tilde{E} \subseteq E$, Let $V' = V \setminus \tilde{V}_O$ and E' be the set of edges with both $\tilde{E}$ and edges incident to $\tilde{V}_O$ removed. Let $\mathcal{G} = (V', E')$ denote the remaining graph. If $O' = O \cap V'$, then let

$$I' = \{u \in I \mid \textit{there exists a path from } u \textit{ to some } v \in O'\}.$$

Then

$$|I'| \geq n\left(1 - \frac{|\tilde{E}|/2}{n - |\tilde{V}_O|}\right).$$

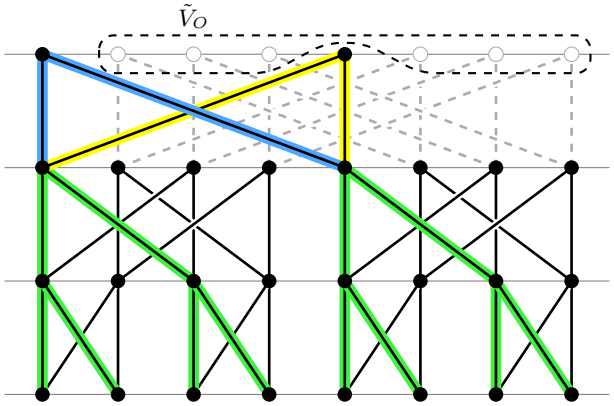


Figure 8: A butterfly network for 8 values with $\tilde{V}_O$ containing 6 of the 8 output nodes. The highlighted edges represent the binary trees rooted at the remaining output nodes.

Proof. First we note that if $|\tilde{V}_O| + \frac{|\tilde{E}|}{2} \geq n$, $\frac{|\tilde{E}|/2}{n-|\tilde{V}_O|} \geq 1$, so we are attempting to bound $|\tilde{I}_v|$ below by a non-positive value. This is trivially true. We will now assume $|\tilde{V}_O| + \frac{|\tilde{E}|}{2} < n$.

$\mathcal{B}_k$ has $k+1$ layers each with n nodes. For each $v \in O$, we can view v as the root of a binary tree B_v whose leaves are the input nodes I.[6] As these trees are overlapping, we note that $V = \bigcup_{v \in O} B_v$.

We start by considering the case where $|O'| = 2^a$. In this case, an optimal positioning of the remaining output nodes causes maximal overlap of their respective binary trees. In particular, if we consider the input nodes to be on layer 0 of the trees and the outputs on layer k, then it is possible for all trees to overlap beginning at layer $k-a$. For example, Figure 8 depicts a maximally-overlapping position when $|O'| = 2$ and thus $a = 1$. We further note that any edges not contained in any of these trees (that is $E \setminus \left(\bigcup_{v \in O'} B_v\right)$) are completely irrelevant; they sit along no paths from any input to any output in O', so there is no reason to include them in $\tilde{E}$. In Figure 8 these irrelevant edges are not highlighted, while those that may be relevant in $\tilde{E}$ have some highlight.

We refer to an edge as being at layer i if it connects edges at layers i and $i-1$.

To find the maximally-efficient choice of edges for $\tilde{E}$, we consider the edges at layer i for each $i \in [1, k]$. The relevant edges at layer $k-a$ are precisely the edges in $\bigcap_{v \in O'} B_v$. This means that by including such an edge (u, v) in $\tilde{E}$, we are able to disconnect every leaf in the subtree rooted at u (in layer $k-a-1$) from every remaining output node. This means disconnecting 2^{n-a-1} inputs from every output. Including an edge at a lower layer is strictly less efficient as it disconnects fewer leaves.

An edge (u, v) *above* layer $k-a$, however, has a different property: it disconnects more inputs from some outputs, but not all. In particular, if (u, v) is at layer $k-b > k-a$ in the tree, then u roots a subtree of size 2^{k-b-1} inputs and removing the edge disconnects those from 2^b outputs. This selection does not, by itself, change I', as there is some $v \in O'$ it does not affect. However, we can choose similar edges for each such v. If we select only edges at layer $k-b$, we must affect all 2^a remaining output nodes, so we must include at least $2^a/2^b = 2^{a-b}$ edges in $\tilde{E}$. While this selection disconnects at least 2^{k-b-1} inputs from each output in O', so too does selecting 2^{a-b} distinct edges at layer $k-a$:

$$2^{k-a-1} \cdot 2^{a-b} = 2^{k-b-1}.$$

[6] The directed edges in $\mathcal{B}_k$ point towards the root of this tree.

This means that any combination of vertices in $\tilde{E}$ that disconnects at least 2^{k-b-1} inputs from every remaining output must include at least 2^{a-b} elements. Simply by performing the same operation multiple times, we see that it requires $c2^{a-b}$ edges in $\tilde{E}$ to disconnect $c2^{k-b-1}$ inputs from every remaining output.

We now consider the case where $|O'| \neq 2^a$. In this case we can partition O' into disjoint subsets $O_1, \ldots, O_t$ where $|O_i| = 2^{a_i}$ with $a_i \neq a_j$ for $i \neq j$. Note that these a_is are the one bits in the binary representation of $|O'|$, so $\sum_{i=1}^{t} 2^{a_i} = |O'|$. In the best case for O_i, including edges at layer $k - a_i$ in $\tilde{E}$ will maximally disconnect inputs from the nodes in O_i. Moreover, because of the binary nature of the butterfly graph, the nodes in O_i are the *only* output nodes in all of O connected to the relevant edges at layer $k - a_i$. This means that, in order to disconnect nodes in O_i and O_j at the same time, we must utilize a layer lower than $k - a_i$, thus reducing efficiency. Instead we can include in $\tilde{E}$ appropriate edges at layers $k - a_i$ and $k - a_j$ to independently disconnect outputs in O_i and O_j, respectively. To disconnect at least $c2^{k-b-1}$ nodes from every element of O', we must therefore have

$$|\tilde{E}| \geq \sum_{i=1}^{t} c2^{a_i - b} = c2^{-b} \sum_{i=1}^{t} 2^{a_i} = \frac{c}{2^b}|O'|.$$

Note that $|O'| = |O \setminus \tilde{V}_O| = |O| - |\tilde{V}_O| = n - |\tilde{V}_O|$.

Since every input begins connected to every output, to ensure that $|I'| = d$, we require that at least $n - d$ input nodes be disconnected from all nodes in O'. If we let c be an odd number and b be an integer such that $n - d = c2^{k-b-1} = cn/2^{b+1}$, the above argument requires that

$$\frac{|\tilde{E}|/2}{k - |\tilde{V}_O|} \geq \frac{1}{2}\left(\frac{1}{n - |\tilde{V}_O|}\right)\frac{c}{2^b}\left(n - |\tilde{V}_O|\right) = \frac{c}{2^{b+1}}.$$

As $d = n\left(1 - \frac{c}{2^{b+1}}\right)$, this concludes the proof. □

Lemma 2. *Let $s_1, \ldots, s_\ell \in [0, 1)$ such that $\sum_{i=1}^{\ell} s_i < 1$. Let $c_0 = 0$ and $c_i = \frac{s_i}{1 - c_{i-1}}$ for $i \geq 1$. If there is some constant α such that $c_i \geq \alpha$ for all $i \geq 1$, then $\alpha < \frac{1}{\ell - 1}$.*

Proof. We begin this proof with a lemma that we will use repeatedly below.

Sublemma 1. *For all $i \geq 0$,*

$$c_{i+1} < 1 - \sum_{j=1}^{i-1} s_j.$$

Proof. We first claim by induction that for $i \geq 0$

$$c_{i+1} \leq s_{i+1}\left(\frac{1 - \sum_{j=1}^{i-1} s_j}{1 - \sum_{j=1}^{i} s_j}\right).$$

For $i = 0$ both sums are empty, requiring $c_1 \leq s_1 \cdot 1$, which holds since $c_0 = 0$. We now assume by induction that the hypothesis holds for i. Applying this inductive hypothesis to the definition $c_{i+1} = s_{i+1}/(1 - c_i)$

yields

$$\begin{aligned} c_{i+1} &\leq \frac{s_{i+1}}{1 - s_i \left(\frac{1-\sum_{j=1}^{i-2} s_j}{1-\sum_{j=1}^{i-1} s_j} \right)} \\ &= \frac{s_{i+1}}{\left(\frac{1-\sum_{j=1}^{i-1} s_j - s_i(1-\sum_{j=1}^{i-2} s_j)}{1-\sum_{j=1}^{i-1} s_j} \right)} \\ &= s_{i+1} \left(\frac{1 - \sum_{j=1}^{i-1} s_j}{1 - \left(\sum_{j=1}^{i} s_i \right) + s_i \left(\sum_{j=1}^{i-2} s_j \right)} \right). \end{aligned}$$

For simplicity, let $t = s_i \left(\sum_{j=1}^{i-2} s_j \right)$. We note that since $s_j \geq 0$ for all j, $t \geq 0$. This means

$$c_{i+1} \leq s_{i+1} \left(\frac{1 - \sum_{j=1}^{i-1} s_j}{1 - \sum_{j=1}^{i} s_j + t} \right) \leq s_{i+1} \left(\frac{1 - \sum_{j=1}^{i-1} s_j}{1 - \sum_{j=1}^{i} s_j} \right).$$

We can now use this inductive result to prove our sublemma. Since the s terms sum to less than 1, we know that $s_{i+1} < 1 - \sum_{j=1}^{i} s_j$. Plugging this into the above inequality, we see that $c_{i+1} < 1 - \sum_{j=1}^{i-1} s_j$. □

Next we note that $\frac{s_{i+1}}{1-c_i} = c_{i+1} \geq \alpha$ means that $s_{i+1} \geq \alpha(1 - c_i)$. Let $\delta_{i+1} = s_{i+1} - \alpha(1 - c_i) \geq 0$ be the distance from that bound. We now claim that for all $i \geq 1$,

$$s_{i+1} \geq \alpha(1 - \alpha) - \delta_i + \delta_{i+1}.$$

To prove this, note that

$$c_i = \frac{s_i}{1 - c_{i-1}} = \frac{\alpha(1 - c_{i-1}) + \delta_i}{1 - c_{i-1}} = \alpha + \frac{\delta_i}{1 - c_{i-1}}.$$

This means

$$\begin{aligned} s_{i+1} &= \alpha(1 - c_i) + \delta_{i+1} \\ &= \alpha \left(1 - \alpha - \frac{\delta_i}{1 - c_{i-1}} \right) + \delta_{i+1} \\ &= \alpha(1 - \alpha) - \delta_i \left(\frac{\alpha}{1 - c_{i-1}} \right) + \delta_{i+1}. \end{aligned}$$

Therefore we must show that $\alpha/(1 - c_{i-1}) \leq 1$ (equivalently $\alpha \leq 1 - c_{i-1}$ since both values are positive) for all $i \geq 1$. We independently prove this for $i = 1$, $i = 2$, $i = 3$, and $i \geq 4$.

CASE $i = 1$: In this case $c_{i-1} = c_0 = 0$, so we need $\alpha \leq 1$, which must be true since $\alpha \leq c_1 = s_1 < 1$.

CASE $i = 2$: In this case $c_{i-1} = c_1 = s_1 = \alpha + \delta_1$. Thus we aim to show $\alpha < 1 - \alpha - \delta_1$. Since we are bounding s_3, we know $\ell \geq 3$, and by Sublemma 1, $c_3 < 1 - s_1 = 1 - \alpha - \delta_1$. Since $\alpha \leq c_3$, this concludes the case.

CASE $i = 3$: To show $\alpha < 1 - c_2$ we similarly note that we are bounding s_4, so $\ell \geq 4$. By Sublemma 1,

$$\begin{aligned}\alpha \leq c_4 &< 1 - s_1 - s_2 \\ &= 1 - c_1 - (\alpha(1 - c_1) + \delta_2) \\ &= (1 - \alpha)(1 - c_1) - \delta_2.\end{aligned}$$

Dividing by $1 - c_1$, we get

$$\frac{\alpha}{1 - c_1} < 1 - \alpha - \frac{\delta_2}{1 - c_1}.$$

Since $0 < 1 - c_1 \leq 1$, we see that $\alpha \leq \alpha/(1 - c_1)$. This gives us the desired result that

$$\alpha < 1 - \alpha - \frac{\delta_2}{1 - c_1} = 1 - c_2.$$

CASE $i \geq 4$: Finally, we handle the case for all $i \geq 4$. In this case $i - 3 \geq 1$, so Sublemma 1 means $c_{i-1} < 1 - s_1 \leq 1 - \alpha$. By adding α and subtracting c_{i-1} from both sides, we get $\alpha < 1 - c_{i-1}$, as desired.

Thus we see that $s_{i+1} \geq \alpha(1 - \alpha) - \delta_i + \delta_{i+1}$ for all $i \geq 1$. By assumption, the s_i's sum to less than 1, meaning

$$\begin{aligned}1 > \sum_{i=1}^{\ell} s_i &= s_1 + \sum_{i=1}^{\ell-1} s_{i+1} \\ &\geq s_1 + \sum_{i=1}^{\ell-1} \big(\alpha(1 - \alpha) - \delta_i + \delta_{i+1}\big) \\ &= \alpha + \delta_1 + (\ell - 1)\alpha(1 - \alpha) + \sum_{i=1}^{\ell-1} (\delta_{i+1} - \delta_i) \\ &= \alpha + (\ell - 1)\alpha(1 - \alpha) + \delta_\ell.\end{aligned}$$

Since $\delta_\ell \geq 0$, we can simply remove that term. Therefore

$$\begin{aligned}&\alpha + (\ell - 1)\alpha(1 - \alpha) < 1 \\ \iff\ &(\ell - 1)\alpha(1 - \alpha) < 1 - \alpha \\ \iff\ &\alpha < \frac{1}{\ell - 1}.\end{aligned}$$

□

With Lemmas 1 and 2 in hand, we can prove our theorem.

Theorem 2. *For any (γ, δ)-depth-robust graph $\mathcal{G}$ with $n = 2^k$ nodes, the associated DSaG $\widehat{\mathcal{G}}$ is $(1, \delta)$-compression-robust.*

Proof. Let $\widehat{\mathcal{G}}'$ be $\widehat{\mathcal{G}}$ after removing data edges $\tilde{E}_D$ and their corresponding key edges.

At layer i, we model vertices in $\mathcal{G}^i$ as begin removed if they cannot reach any non-removed vertex in $\mathcal{G}^{i+1}$, with vertices removed from $\mathcal{G}^\ell$ if they cannot reach a sink node in $\widehat{\mathcal{G}}'$. As removing the outgoing edge from a vertex in $\mathcal{G}^i$ disconnects it from $\mathcal{B}^i$ and thus all vertices in $\mathcal{G}^{i+1}$, any internal edges in $\mathcal{G}^i$ between non-removed vertices will still be present. This means that any path in $\mathcal{G}^i$ containing only remaining edges, will still be present. Since $\mathcal{G}$ is an (γ, δ)-DRG, it suffices to show that, for some i, we remove fewer than

an γ fraction of the vertices. The depth-robustness then guarantees that there is a path of δn (key) edges remaining in $\mathcal{G}^i$. Because all non-removed vertices in one layer can reach some non-removed vertex in the next, we can connect the terminal vertex of this δn-length path of key edges to a sink node of $\widehat{\mathcal{G}}$.

At layer ℓ, we remove a vertex from $\mathcal{G}^\ell$ if its single output data edge to a sink vertex has been removed. For $i < \ell$, we remove vertices as follows. Let U^{i+1} be the set of non-removed vertices in $\mathcal{G}^{i+1}$. We let U^i be the set of vertices in $\mathcal{G}^i$ from which some vertex in U^{i+1} is reachable in $\widehat{\mathcal{G}}'$.

As argued above, it suffices to show that $|U^i|/n > 1 - \gamma$ for some $i \in [1, \ell]$.

We first consider $i < \ell$. Let V_O^i be the set of output vertices from $\mathcal{B}^i$ neither of whose children are in U^{i+1}. That is, V_O^i is the set of outputs from $\mathcal{B}^i$ that cannot reach any vertex in U^{i+1}, and thus may not be able to reach a sink vertex of $\widehat{\mathcal{G}}'$. As each output of $\mathcal{B}^i$ connects to a pair of vertices in $\mathcal{G}^i$ and no two pairs overlap, $|V_O^i| \leq \frac{1}{2}|U^{i+1}|$. Further, let $\tilde{E}^i = \tilde{E}_D \cap \mathcal{B}^i$ be the edges removed from $\mathcal{B}^i$. As in Lemma 1, let I'^i be the set of input nodes in $\mathcal{B}^i$ that can reach some output node outside V_O^i. By Lemma 1,

$$|I'^i| \geq \frac{n}{2}\left(1 - \frac{|\tilde{E}^i|/2}{n/2 - |V_O^i|}\right).$$

Each vertex I'^i connects to some vertex in U^{i+1} and has two unique parents in $\mathcal{G}^8$, so

$$|U^i| \geq n\left(1 - \frac{|\tilde{E}^i|/2}{n/2 - |V_O^i|}\right) \geq n\left(1 - \frac{|\tilde{E}^i|}{n - |U^{i+1}|}\right).$$

We can normalize by dividing all set by n, so for $i < \ell$,

$$s_i = |\tilde{E}^i|/n \quad \text{and} \quad u_i = |U^i|/n.$$

This means $u_i \geq 1 - \frac{s_i}{1-u_{i+1}}$. We additionally let $\tilde{E}^\ell$ denote the set of edges in $\tilde{E}_D$ between $\mathcal{G}^\ell$ and the sink nodes of $\widehat{\mathcal{G}}$, and $s_\ell = |\tilde{E}^\ell|/n$ accordingly. Since $\sum_{i=1}^{\ell} |\tilde{E}^i| \leq |\tilde{E}_D| < n$, we know that $\sum_{i=1}^{\ell} s_i < 1$.

We can now reverse the ordering of the numbering, letting $s'_i = s_{\ell-i+1}$, and $c_i = 1 - u_{\ell-i+1}$. This means that $\sum s'_i < 1$ and $c_i = \frac{s'_i}{1-c_{i-1}}$. Therefore Lemma 2 say that if there is some α such that $c_i \geq \alpha$ for all $i \in [1, \ell]$, then $\alpha < \frac{1}{\ell-1}$. In particular, we know that $\ell \geq \frac{1}{\gamma} + 1$, and therefore $\alpha < \gamma$. This means there is some $i \in [1, \ell]$ such that $c_i < \gamma$. If we let $i' = \ell - i + 1$, then $u_{i'} = |U^{i'}|/n > 1 - \gamma$. □