Hourglass Schemes: How to Prove that Cloud Files Are Encrypted

Marten van Dijk
RSA Laboratories
Cambridge MA
marten.vandijk@rsa.com

Ari Juels
RSA Laboratories
Cambridge MA
ari.juels@rsa.com

Alina Oprea
RSA Laboratories
Cambridge MA
alina.oprea@rsa.com

Ronald L. Rivest
MIT
Cambridge MA
rivest@mit.edu

Emil Stefanov
UC Berkeley
Berkeley CA
emil@berkeley.edu

Nikos Triandopoulos
RSA Laboratories
Cambridge MA
nikolaos.triandopoulos@rsa.com

ABSTRACT

We consider the following challenge: How can a cloud storage provider prove to a tenant that it's encrypting files at rest, when the provider itself holds the corresponding encryption keys? Such proofs demonstrate sound encryption policies and file confidentiality. (Cheating, cost-cutting, or misconfigured providers may bypass the computation/management burdens of encryption and store plaintext only.)

To address this problem, we propose *hourglass schemes*, protocols that prove correct encryption of files at rest by imposing a resource requirement (e.g., time, storage or computation) on the process of translating files from one encoding domain (i.e., plaintext) to a different, target domain (i.e., ciphertext). Our more practical hourglass schemes exploit common cloud infrastructure characteristics, such as limited file-system parallelism and the use of rotational hard drives for at-rest files. For files of modest size, we describe an hourglass scheme that exploits trapdoor one-way permutations to prove correct file encryption whatever the underlying storage medium.

We also experimentally validate the practicality of our proposed schemes, the fastest of which incurs minimal overhead beyond the cost of encryption. As we show, hourglass schemes can be used to verify properties other than correct encryption, e.g., embedding of "provenance tags" in files for tracing the source of leaked files. Of course, even if a provider is correctly storing a file as ciphertext, it could *also* store a plaintext copy to service tenant requests more efficiently. Hourglass schemes cannot guarantee ciphertext-only storage, a problem inherent when the cloud manages keys. By means of experiments in Amazon EC2, however, we demonstrate that hourglass schemes provide strong incentives for economically rational cloud providers against storage of extra plaintext file copies.

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

Categories and Subject Descriptors

C.2.4 [**Communication Networks**]: Distributed Systems—*Client/server*; E.3 [**Data Encryption**]; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms

Algorithms, Security, Theory, Verification

Keywords

cloud storage security, cloud auditing, challenge-response protocol, economic security model

1. INTRODUCTION

Uncontrolled data leakage is an enormous problem today. It costs the industry billions of dollars of damage each year and attracts significant media attention with high-profile incidents such as [2, 20]. This problem is further amplified as cloud storage becomes increasingly popular. Cloud data is often subject to a larger number of attack vectors and the responsibility of securely managing this data is split across multiple parties.

Today's cloud providers implement complex distributed storage and caching systems, but usually offer this service as a simple API for their tenants. A negative side-effect is that these cloud storage APIs do not provide tenants with the necessary degree of transparency for them to verify that data is properly secured, e.g., via encryption. They simply have to trust that the cloud provider is "doing the right thing." Unfortunately, history has demonstrated that parties often overlook even basic security measures without strong up-front motivation [1]. It is only after a serious incident occurs, that security measures are implemented, but even then it is not possible for users to verify that they were implemented correctly because such security measures typically lack visibility.

To strongly protect against unauthorized access or disclosure, one option is for clients to always encrypt their data before uploading it to the cloud and decrypt it on-the-fly when needed, using their own keys that are kept secret from the cloud. While this option may work for some scenarios, it is too restrictive in many other cases as it undermines much of the benefit of outsourcing to the cloud. Clouds are

often used for more than just plain storage: To provide more essential services, it's very common for clouds to require access to the unencrypted data (e.g., for processing the data or simply furnishing it as plaintext to authorized users).

This then necessitates that data encryption is enforced by the cloud provider itself. Knowing the encryption/decryption keys enables the provider to offer services of better functionality and also minimizes the security burden on tenants that lack security sophistication. But as cloud providers now assume responsibility for data confidentiality they must always keep data at rest encrypted, and protect the encryption keys. However, such data-management cloud services are often proprietary software developed by the providers themselves, thus lacking the necessary transparency for tenants to verify that their private data is handled securely. This renders the above cloud-centric approach preferable in terms of usability, but at the same time less desirable in terms of trustworthiness as it only provides *the promise but no guarantees* for data protection. There is no assurance that cloud data will not leak.

In such settings of cloud-managed encrypted data, it seems that very little can be done to provide transparency to tenants about the correct handling by the cloud providers of their own sensitive data.

In this paper, we demonstrate that this is actually not the case. In fact, it is possible to design a storage protocol that imposes a strong economical incentive onto the cloud provider to *store data securely encrypted at rest*. With our protocol, tenants can remotely *verify* that data in the cloud is encrypted at rest. We ensure that a negligent cloud provider that wishes to store the data unencrypted has a much higher operating cost: Essentially, this provider will need to double its storage capacity and the underlying infrastructure that provides access to that storage in order to pass the verification protocol.

In particular, we present a general framework for economically motivating a cloud provider *to transform and store data F outsourced by a tenant into a particular encoding G of tenant's choice*. To achieve this we introduce a new primitive that we call an *hourglass scheme* that essentially *seals the data immediately after* the data has been encoded in the format requested by the tenant. This sealing is performed by applying a special transformation that we call an *hourglass function* over the encoded format G to get an encapsulation of it in a new format H that is finally stored by the provider. Our framework includes a challenge-response protocol that a tenant can use to *verify that the cloud provider is storing the data with this desired particular encoding G and hourglass function applied to it* (i.e., format H). This function is specifically designed to ensure that the cloud provider cannot apply a subsequent encoding of their choice to reverse the initial one (i.e., format G). The hourglass function is also designed to impose significant resource constraints—and hence an economical disincentive—on a cloud provider that tries to apply the encoding and hourglass functions on demand (on the raw data F) when the client initiates the challenge-response verification protocol.

Specifically, we are able to show that *under the assumption that cloud providers are economically rational*, our hourglass schemes provide strong *disincentives* for direct leakage of "raw" data through a *double storage* dilemma. Given that an hourglass scheme is in use to verify correct encryption of data at rest at all times, retention of encrypted data only is the *rational* strategy for economically motivated providers. Indeed, an economically rational provider who wishes to *also* store unencrypted data needs to essentially double its storage cost. We actually experimentally demonstrate that in many settings of interest, it is *less expensive* for a provider to comply fully with an hourglass scheme and store only encrypted data than to cheat and store an additional, unecrypted copy of the data.

Contributions. Overall, our work has several contributions, both theoretical and practical:

- We are among the first to explore economic security models for cloud services that provide incentives to cloud providers to implement security measures correctly.
- We introduce hourglass schemes as a new solution concept for ensuring that cloud-managed data is securely protected via encryption and formalize a general framework for the design of a class of such schemes.
- We propose several concrete hourglass schemes that impose a strong economic incentive on cloud storage providers to store data properly encrypted at rest.
- We extend our constructions to incentivize the cloud provider to embed watermarks in data files so that the origin of a data leak can be traced back.
- We implement our constructions and experimentally quantify the monetary cost induced on a misbehaving or negligent cloud provider.

Paper organization. We introduce hourglass schemes and present our new framework in Section 2. Our main protocol is presented in Section 3 and is general enough to support multiple kinds of hourglass and encoding functions, lending itself to a generic method for the design of hourglass schemes. In Section 4, we describe three specific hourglass functions that we call the *butterfly*, *permutation* and *RSA* constructions, where the first two are time-based imposing resource constraints on a misbehaving cloud provider that are related to storage access time whereas the third imposes constraints related to computation, and we present their security properties. In Section 5, we explore the effectiveness of our hourglass schemes against an economically motivated cloud provider and validate our economic arguments experimentally by implementing our two time-based hourglass schemes in Amazon's EC2 service. Finally, we review related work in Section 6 and conclude in Section 7. Additional technical materials on our formal security model and protocols for proving correct encoding appear in the Appendix. This extended abstract omits proofs of security which appear in the full version of our paper.

2. GENERAL FRAMEWORK

The overall goal of our paper is to strongly motivate cloud providers to properly protect client data to minimize the damage when an accidental data leakage occurs. We achieve this through a general framework that involves the design of protocols that ensure that client data is stored at rest in an encoding of client's choice. Example encodings supported by our framework include *encryption of data* with keys managed by the cloud provider, and encoding with an *embedded*

watermark that enables tracking back the origin of a data leak. Both of these provide protection against data leakage. To elaborate: encryption of data at rest preserves confidentiality in case of accidental data leakage; watermarked encoding reveals the source of data leakage (more specifically, the identity of the cloud provider), further incentivizing the cloud provider to strongly protect data and implement security measures correctly.

Our framework introduces a new solution concept for ensuring that cloud data resides in a given correct and desired format. It enables the client to specify an encoding of its choice and then *remotely verify* using an efficient challenge-response protocol that its data is stored at rest in the cloud in the desired format. Since the cloud provider in our setting has access to raw data, it can also store (parts of) unencoded data. While we can not completely prevent this situation, our framework provides strong *economic incentives* for the cloud provider to store data only in the encoded format: A misbehaving cloud provider storing raw data would have to *double* its storage cost in order to reply correctly to client challenges.

To illustrate the challenges of building a framework with these requirements, consider the problem of verifying that a certain file F is stored in an encrypted format G by the cloud provider—the main use case in this paper. The client might challenge the cloud provider for some randomly selected blocks of the encrypted file G (or even the full encrypted file G). Demonstrating knowledge of G by itself does not prove that G is stored by the cloud provider. Indeed, the cloud provider could store the plaintext file F, and compute the encryption G on-the-fly when challenged by the client! For most symmetric-key encryption schemes (e.g., block-chaining modes of a block cipher), encryption is a fast operation, and thus on-the-fly computation is feasible.

2.1 Solution overview

To overcome these challenges, we propose that an additional transformation, called an *hourglass function*, is applied to the encrypted file G to get an *hourglass format* H that encapsulates G, and that the client challenges the provider on this new file format H.

An hourglass is an ancient timetelling device. It consists of two glass bulbs connected by a narrow neck and partially filled with sand. In order for sand accumulated in one half to move to the other, it must pass through the neck—a rate-limited process that can measure the passage of time. Similarly, an hourglass function imposes a *resource constraint* (e.g., time) on the translation of a message from one encoding domain to another, target domain.

In particular, such a scheme might ensure a certain lower bound τ on the time required to translate a ciphertext file G into its encapsulation file H. The client can then challenge the cloud provider at a *random* time to produce *random* chunks of the encapsulated file H, and require that the cloud provider do so in time $< \tau$. By successfully complying, the cloud provider proves that *it has actually stored the hourglass file H that encapsulates G (from which G can be efficiently extracted), and is not encrypting it on-the-fly.*

Moreover, the hourglass function must be carefully crafted to ensure that it is not possible to store partial information (e.g., 20% of the file size) to enable efficiently recovering random chunks of the output. In other words, the hourglass function should be applied to the encrypted file G as a whole, and it should be equally difficult or costly to evaluate on portions of the encrypted file as it is to evaluate on the whole file. Existing modes of encryption such as block chaining fail to meet this criteria. Another important requirement is that recovery of the ciphertext G (and, in turn, recovery of plaintext F) from the encapsulation H should be a reasonably efficient operation, in case raw data is needed by the cloud for processing.

Our general framework combines in a challenge-response protocol two basic components, a properly parameterized *resource bound* with a complementary *hourglass function* that exploits this bound:

- **A resource bound**: An hourglass scheme assumes a bound on some resource at the cloud side—storage, computation, or networking. For example, hard drives experience delays due to seek time (time to access a data block) and bandwidth (rate at which they can read out data)—as they are the predominant storage device in storage infrastructures today, they provide one possible foundation for time-based protocols.

- **An hourglass function**: An hourglass function involves an invertible transformation which imposes a lower bound on the resources required to translate between coding domains, i.e., makes translation moderately hard.

An example hourglass scheme. To gain intuition and assuming a computational resource bound on the server, we describe a simple, but somewhat impractical hourglass scheme based on inversion of a small-image hash function. Let F consist of n blocks, $F_1, F_2, \ldots, F_n$, each of l bits. Suppose that the client wishes to verify that the server applies format-preserving encryption to F, to obtain a ciphertext G. Let $h : \{0,1\}^* \to \{0,1\}^l$ denote a fixed-length hash function.

> *Hourglass function.* Consider then an hourglass function defined as follows. Transform G into H by *inverting each block of G under h*; the server computes and stores a pre-image $H_i = h^{-1}(G_i)$ for every $i \in \{1, \ldots, n\}$.[1]

A well-behaving server, then, stores the encapsulated ciphertext H (not the ciphertext G, nor, of course, the plaintext F). Note that recovery of G (and then F) from H is very practical for this hourglass function, by simply computing the hash values $G_i = h(H_i)$ for $i \in [1, n]$.

Verification of storage of H in the hourglass scheme here will rely on the fact that even for fairly small values of l (e.g., 35-bit or 40-bit blocks), the transformation from G_i to H_i is computationally costly on average. In particular, we might appeal to a simple, concrete resource bound, such as the following:

> *Resource bound.* The server can perform at most 2^{l-1} executions of h in time τ.

To verify that a server stores H, then, the client simply selects a random block index i, and challenges the server to furnish H_i. An honest server that has stored H just retrieves and serves H_i—a quick operation. But a cheating server that

[1]Obviously, the server tries to find the shortest possible such pre-image, which will usually be of length close to l.

has stored only the plaintext file F, has to compute H_i on the fly. It must compute ciphertext block G_i and then perform a costly computation of H_i which, on average, takes time τ. In other words, because inversion of h is a moderately hard computational task, it will delay the response of a cheating server, by τ on average. A client can therefore use the response time, r, to distinguish an honest server ($r < \tau$) from a dishonest one ($r \geq \tau$).

While feasible, this hash-function-based hourglass function has some unattractive properties. First, a pre-image H_i will, on average, be a little larger than its image G_i, resulting in some degree of message expansion in the transformation from G to H. Additionally, the full transformation from G to H is quite costly, as it requires n inversions of h. And a cheating server can easily parallelize the inversion of h.

The function h used here is a (compressed-range) one-way function. Later in the paper, we consider the use of a *trapdoor* one-way function, which removes these various drawbacks.

Alternative approaches. Trusted computing offers an alternative approach to address the problem of verifying that data at rest is encrypted. With trusted computing enabled hardware, the client could require that the cloud provider place all of its code that needs to access unencrypted data inside of a trusted computing environment so that the client can remotely attest that the code is correctly encrypting data at rest. The main problem with this approach is that trusted computing can easily be circumvented because the cloud provider has physical access to the machines. Additionally, it would be difficult and costly for a cloud provider to move all of their cloud services into a trusted environment, especially since a large portion of their existing hardware might not be capable of ensuring trusted execution.

2.2 Framework description

Let n denote the number of file blocks in a target file F, where each block belongs in a domain B. We refer to this original format as *raw*. We let F_i denote block i of F, and l' denote the length (in bits) of a file block, i.e., $F_i \in B = \{0,1\}^{l'}$. We let $G \in L^n$ be the block-by-block encoded file, where blocks belong in a domain L, and let l denote the length (in bits) of an encoded file block, i.e., $G_i \in L = \{0,1\}^{l}$. For simplicity, we also let l be the security parameter for our system (typically $l = 128$).

Framework encoding. First, the tenant specifies as input to the hourglass scheme *an encoding algorithm* defined by three operations:

- $\mathsf{keygen\text{-}enc} \xrightarrow{R} \kappa_1$: A key-generation function that outputs a secret key κ_1. To support publicly computable encodings, κ_1 could be the null string.
- $\mathsf{encode}(\kappa_1, F) \rightarrow G$: An encoding mapping that takes a raw file $F \in B^n$ and transforms it block by block into an *encoded* file $G \in L^n$. We assume, as detailed in Appendix B, that G is uniformly distributed in L^n.
- $\mathsf{decode}(\kappa_1, G) \rightarrow F$: A decoding mapping that takes an encoded file $G \in L^n$ and transforms it into a raw file $F \in B^n$.

While our framework is general in supporting arbitrary encoding algorithms, we detail three specific encodings in Appendix A.

Framework operations. Then, *an hourglass scheme* HG is employed that consists of the following set of operations:

- An encoding algorithm given by keygen-enc, encode, decode.
- $\mathsf{keygen\text{-}hg} \xrightarrow{R} (\kappa_2, \kappa_3)$: A key generation function that yields l-bit keys $\kappa_2, \kappa_3 \in L$, where κ_3 is optional (only used in our RSA-based construction in Section 4.3). If κ_3 is not used in a particular construction, its value is set to the null string.
- $\mathsf{hourglass}(\kappa_2, \kappa_3, G) \rightarrow H$: An hourglass function encapsulates an encoded file $G \in L^n$ in a format suitable for proof of correct application of encode. The encapsulation produces $H \in D^n$ (where in general $|D| \geq |L|$). The function hourglass may be deterministic or probabilistic.
- $\mathsf{reverse\text{-}hourglass}(\kappa_2, H) \rightarrow G$: The inverse of the hourglass function reverses hourglass, reverting a storage-formatted file $H \in D^n$ back to an encoded file $G \in L^n$.
- $\mathsf{challenge} \xrightarrow{R} c$: A function that generates a challenge. For simplicity, we assume a randomly selected block index $c \in \{1, \ldots, n\}$ of H.
- $\mathsf{respond}(H, c) \rightarrow r$: An algorithm that operates over an hourglass-encapsulated file $H \in D^n$ to respond to a challenge $c \in \{1, \ldots, n\}$. For simplicity, we assume $r \in D$.
- $\mathsf{verify}(H, c, r)$: Verifies the response to a challenge $c \in \{1, \ldots, n\}$. For simplicity, we assume that the verifier knows H. In practice, of course, this requirement can be easily relaxed by, e.g., having the verifier MAC blocks of H or maintain a Merkle tree over blocks of H (i.e., an authentication tree which itself can be outsourced to the cloud).

We say that an hourglass scheme HG is *valid* if for all keys produced as output by $\mathsf{keygen\text{-}enc} \xrightarrow{R} \kappa_1$ and $\mathsf{keygen\text{-}hg} \xrightarrow{R} (\kappa_2, \kappa_3)$ the following three conditions hold with overwhelming probability in l:

1. $\mathsf{decode}(\kappa_1, \mathsf{encode}(\kappa_1, F)) = F$;
2. $\mathsf{reverse\text{-}hourglass}(\kappa_2, \mathsf{hourglass}(\kappa_2, \kappa_3, G)) = G$;
3. For all $c \leftarrow \mathsf{challenge}$, $\mathsf{verify}(H, c, \mathsf{respond}(H, c)) = 1$.

Note that the second condition above implies that G *can be efficiently extracted from the encapsulated file* H, i.e., G can be recovered from H. Thus, verification (by the client) of challenged blocks of H implies that H is stored (by the cloud) and that, thus, G is recoverable.

Appendix B presents the formal security definitions for hourglass schemes. While an honest server will store representation H in its storage, intuitively, the goal of an adversarial server is to construct storage H' with two properties: (1) The adversary can respond to a large fraction of client challenges by accessing blocks of H'; and (2) The adversary leaks parts of the plaintext file F in H'. We make a simplifying assumption about the adversary in the security analysis of the three constructions described in Section 4. This *partitioning assumption* detailed in Appendix B requires that the

adversary separates its storage H' into $H' = (H'_F \parallel H'_G)$, where H'_F is derived from the plaintext file F (representing what the adversary leaks about file F) and H'_G is computed over G (and used to answer client challenges). In our theorem statements in Section 4 we give lower bounds on the amount of extra storage $s' = |H_G|$ imposed on a cheating server trying to leak part of the plaintext file and to achieve success probability α in the challenge-response protocol.

We give three concrete instantiations of hourglass functions in Section 4 along with their security analysis. (These are based on either storage bounds—in particular bounds on hard drive access time and throughput—or computational bounds). But first, we next show how an encoding algorithm and an hourglass function with appropriate properties can be combined in a generic protocol, thus overall supporting a rich class of hourglass schemes.

3. GENERIC HOURGLASS PROTOCOL

To review our discussion thus far, the goal of an hourglass protocol is to provide assurance to a client that its data outsourced to a server is stored in a desired, encoded format G. The encoded format is computed by applying the encode algorithm to the original file F. For most file encodings of interest (e.g., encryption, watermarking), the output of encode is fast to compute given file F. As such, the client cannot challenge the server directly on encoding G to ensure the application of encode to F: The server can store file F and compute G on-the-fly when challenged!

To resolve this issue, an hourglass protocol encapsulates the encoding G into another format H suitable for remote format verification. The transformation from G to H is performed with an hourglass function that enforces a resource bound (e.g., minimum amount of time, storage or computation). This function is easily reversible: Knowledge of H implies that G can be easily retrieved and, in turn, that blocks of the original file F can only be obtained with access to the decode algorithm. To ensure storage of format H, the client challenges the server on randomly selected blocks of H. By the correctness and speed of the response, the client gets guarantees about the fact that the server stores the file in hourglass-encapsulated format H.

Accordingly, our framework considers a generic hourglass protocol that consists of three phases. Phase 1 performs the *file encoding*, where the file F is encoded into G by the server and a *proof of correct encoding* is generated and sent back to the client. Phase 2 performs the *hourglass encapsulation*, where the hourglass transformation is applied to G to obtain file H, stored by the server. Finally, Phase 3 performs *format checking*, where the client checks that the server indeed stores H by challenging the server with randomly chosen blocks of H. The first two phases are executed only once, whereas the third one is executed periodically, in unpredicted (e.g., randomized) intervals. Below we elaborate on each one of the three phases of our generic hourglass protocol which is formally presented in Figure 1.

3.1 Phase 1: file encoding

A generic hourglass protocol performs the initial transformation from the original file F provided by the client to the target encoding G using the encode algorithm. Given G, the original file F can be retrieved using the corresponding decode algorithm. In particular, in Phase 1 the server applies encode to the original file F (received from the client) to obtain G. The encoding G, as well as a proof of correct encoding, is sent to the client—G is needed by the client in order to apply the hourglass transformation in Phase 2. At the end of this phase, the client is assured (with high probability) that encode has been applied correctly to F.

Overall, by appropriately instantiating this pair of algorithms (encode, decode), our framework can support protocols for ensuring storage of the following three file encodings (these are detailed in Appendix A):

1. Encryption: We present a protocol by which a client (or third party, e.g., auditor) can verify that a storage provider stores a file F in encrypted form. As mentioned before, we consider a natural (and technically challenging) model in which the provider manages the encryption key, and never divulges it to the verifier. (This protocol is the main focus of the paper.)

2. Watermarking: We show how a server can prove that a file F is encoded with an embedded provenance tag τ: If F is leaked, τ identifies the server as the source, and thus the responsible/liable entity. We describe an encoding such that it is infeasible to learn F without learning τ, i.e., learning the file implies learning its provenance tag. The major challenge in implementing an hourglass scheme for this encoding is enabling the server to prove to the client that τ is embedded in F without revealing τ itself. This is important, as a client (or third party) that learns τ could frame the storage provider, falsely furnishing τ as evidence that F has leaked.

3. File bindings: We show how a server can prove that a pair of files (F_1, F_2) are stored in such a way that retrieval of one file implies retrieval of the other. For instance, F_1 might be a piece of software and F_2 an accompanying license agreement: Binding the two together ensures that any entity retrieving F_1 also gets F_2 and thus cannot claim failure to receive critical legal information governing software usage.

Above we highlight only three concrete encodings that are useful in practice, but these applications are by no means exhaustive. They just represent examples of file encodings for which we can construct hourglass protocols. We believe an interesting area of research is finding other classes of encodings that could be embedded into an hourglass protocol.

Note that this phase involves more than just outsourcing of file F to the server. While function hourglass is always computable by the client, function encode is not always available to the client.[2] We thus remove this obstacle by requiring the server to send in this phase G to the client *along with a proof that G is a correct encoding*. Once this proof is verified, the client can then apply in the next phase the hourglass transformation on G to obtain H. These protocols appear in Appendix A.

3.2 Phase 2: hourglass encapsulation

The hourglass protocol additionally performs the transformation from G to the hourglass format H that involves applying the hourglass function hourglass on the encoded file G. In particular, in Phase 2 the client and, in some cases,

[2]Recall that in the case of proving correct at-rest file encryption, the client does not know the file encryption key.

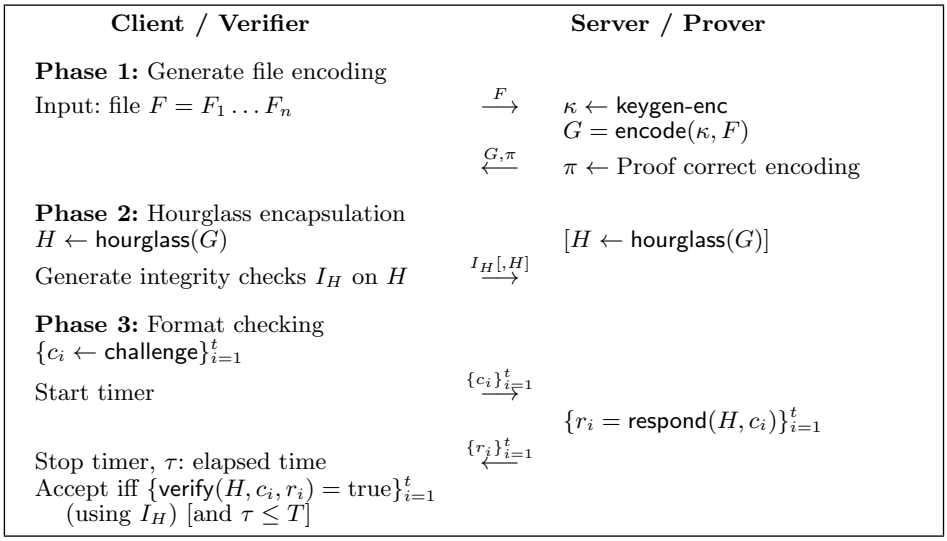


Figure 1: Generic hourglass protocol

the server applies hourglass to encoding G to compute the encapsulation H for storage on the server. This is a block-by-block transformation that produces n hourglass blocks $H_1, \dots, H_n$. Note that if hourglass does not use a secret key, then both parties can compute H directly from G. This observation saves one round of n-block communication in our butterfly and permutation constructions from Section 4.

An issue arising in the design of an hourglass protocol is the fact that the client needs to verify responses from the server in the challenge-response below (Phase 3). In our system definition we assume for simplicity that H is available to the verifier (algorithm $\mathsf{verify}(H, c, r)$ explicitly receives H as input). In a practical implementation of an hourglass protocol—particularly in cloud applications—it's desirable to have H stored only by the server.[3] To verify a block of H (returned by the server as a response to a challenge), the client can pre-compute integrity checks for blocks of H. This requires that the client itself possess the authentic H.

3.3 Phase 3: format checking

The hourglass protocol includes a final challenge-response component (executed multiple times at unpredictable time intervals) in which the client can check that the server is indeed storing the file in format H. In particular, in Phase 3 the client challenges the server to verify that the server in fact stores H. In the challenge protocol the client chooses t random block indices $\{c_i\}_{i=1}^t$ and challenges the server to produce responses $\{r_i\}_{i=1}^t$ which are sent to the client.

The client then proceeds with the verification of the received responses by first simply checking their correctness with respect to the authentic blocks of H (originally produced in Phase 2). This is achieved by making use of the integrity checks on the received blocks of H (as discussed above). In addition, in our time-based constructions, the client measures the time between its sending the challenge and receiving a reply. It accepts the response if it arrives in time less than some security parameter T. We note that as an optimization, the server can optionally aggregate responses before sending them to the client (as, e.g., in [27]).

We emphasize that the client's integrity-checking mechanism is orthogonal to the rest of our hourglass protocols. The client can use any standard scheme that supports verification of blocks of H, e.g., MACs or Merkle trees. Once generated by the client, integrity checks can be stored by the client, or preferably stored on the server and retrieved in the challenge-response protocol. If MACs are used, the client retains any secret keys required for verification; if Merkle trees are used, the client needs store no secret state but just the root of the tree. For the remainder of the paper, we assume the use of an integrity checking scheme and omit details.

4. HOURGLASS FUNCTIONS

We now present our concrete hourglass functions. To illustrate the challenge of constructing good hourglass functions, we first present a naïve approach and briefly explain why it doesn't work. Then we explore three constructions, namely:

1. A *butterfly* hourglass function in Section 4.1, that corrects the flaw in the naïve approach. This approach is timing based and assumes general bounds on storage speed (that as we will discuss typically hold in all existing storage media).

2. A *permutation-based* hourglass function in Section 4.2. This timing-based scheme assumes a more specific storage model: It relies on particular properties of rotational hard-drives, namely their performance gap in accessing sequential disk sectors versus randomly selected ones (whose access induce an expensive drive seek operation). Using non-cryptographic data-word permutation, this construction is very efficient.

3. An *RSA-based* hourglass function in Section 4.3. This scheme involves application of an RSA signing operation (repeatedly for strong security properties) to individual file blocks. It's less efficient than the other two

[3]Ideally, the client's storage needs should be of constant-size (independent of the file size n); otherwise the benefits of data outsourcing diminish.

schemes, but has the advantage of relying on the hardness of the RSA problem, rather than timing, and it supports random access to files as well as file updates.

We analyse the security guarantees for all of our proposed constructions above (and we provide complete security proofs in the full version of this paper). We experimentally explore their performance in Section 5.

Challenges. Recall that both the encoded file G and the hourglass transformation H in our framework are represented as n-block files, where each block has fixed size. Our first two time-based hourglass functions that exploit delays on data-access speed rely on a common underlying approach. They transform G into H so that every block H_i functionally depends on a large set S_i of blocks in G. Intuitively, an hourglass function that achieves good "mixing" of this sort seems to offer good security. Suppose that a server cheats by storing F instead of H. When challenged to produce H_i, the server would have to retrieve all the blocks in S_i. By our specific bounds on data retrieval speed this task is slow imposing a noticeable response delay.

We might then consider a simple and efficient hourglass function that "mixes" optimally, in the sense that every block H_i depends on *all of* G. This is achievable with a full-file (known-key) pseudorandom permutation (PRP) over G, computable with two cipher-block chaining (CBC) passes over the file, one in the forward direction and one in the reverse direction: Then every block of H depends on all blocks of G.

Specifically, let IV_f and IV_b be initialization vectors and $\mathsf{Enc}_\kappa(\cdot)$ denote encryption under a block cipher. (The key κ should be known to the server so that it can reverse the PRP: This doesn't impact the cryptographic effect of "mixing.") The forward pass computes intermediate values $A_1 = \mathsf{Enc}_\kappa(G_1 \oplus IV_f)$, $A_i = \mathsf{Enc}_\kappa(G_i \oplus A_{i-1})$, for $1 < i \leq n$; the backward pass computes output blocks $H_n = \mathsf{Enc}_\kappa(A_n \oplus IV_b)$, and $H_i = \mathsf{Enc}_\kappa(A_i \oplus H_{i+1})$, for $1 \leq i < n$.

As it turns out, though, because this double-pass CBC function involves chaining of blocks, it is possible for a cheating server to store F and a "shortcut," namely staggered, intermediate blocks within a chain. In particular, let $H' = \{H_v, H_{2v}, \ldots H_{v \times \lfloor n/v \rfloor}\}$ for some parameter v. These intermediate blocks allow the server to compute blocks of H efficiently on the fly from a compact H'. For small enough v, the server could quickly compute a challenge block H_i (for $jv \leq i \leq (j+1)v$) by computing the forward chain segment $A_{jv}, \ldots A_{(j+1)v}$ to retrieve $H_{(j+1)v}$, and then computing the backward chain segment $H_{(j+1)v}, H_{(j+1)v-1}, \ldots, H_i$. Of course, a small value of v induces a large amount of extra storage, thus parameter v needs to be chosen to balance the storage overhead of intermediate blocks, as well as the cost of computing responses to challenges on-the-fly.

This approach fails because a single block A_j: (1) Sits on the path of computation for many output blocks in H and (2) Allows a server with knowledge of A_j (along with G) to compress substantially the amount of computation along many of these paths. Good mixing is thus a necessary but not sufficient security condition.

This example illustrates the subtle challenges in constructing a good hourglass function, and the motivation for our next two hourglass function constructions.

4.1 A butterfly hourglass function

We now propose what we call a *butterfly* hourglass function. It replaces the flawed, chained-block dependencies of the naïve double-pass CBC construction that we saw above, with a structure of overlapping binary trees. At the same time, this function is a full-file PRP, and so achieves the security requirement of strong "mixing." Storage of intermediate results in a butterfly provides little benefit to a cheating server: As we explain, a server can't effectively compress computational paths from G to H into a "shortcut."

A butterfly function applies an (atomic) cryptographic operation w to pairs of blocks in a sequence of $d = \log_2 n$ rounds, as shown in Figure 2.[4] The resulting structure of pairwise operations resembles a *butterfly network* commonly used for parallelized sorting, FFT, etc.

Let G and H be n-block files, with each block having size l bits (that is, $|L| = |D| = l$). Let $w : L \times L \leftrightarrow L \times L$ denote the atomic operation over a pair of file blocks. In practice, we can instantiate w as a (known-key) PRP, i.e., a block cipher. Formally, $\mathsf{hourglass} : L^n \leftrightarrow L^n$ computes in d rounds a transformation from n-block input file G to output file H as follows. Define $G_0[i] = G_i$ for all i. For $1 \leq j \leq d$, we compute the output $G_j[1] \ldots G_j[n]$ of level j as a function of level $j-1$, as specified in Algorithm 1 of Figure 2.

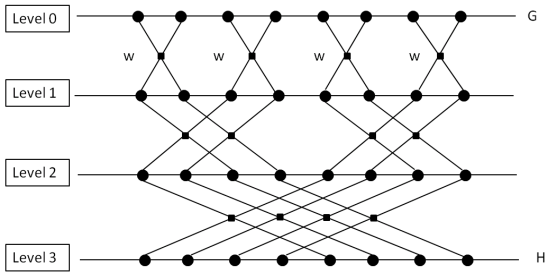
In Figure 2, we present an example butterfly network for the case $n = 8$. In this representation, each set of values $G_j[1], \ldots, G_j[n]$ is a row of nodes. Two edges connect the input and output node pairs involved in each application of w, where w itself is depicted as a small square. Globally, $\mathsf{hourglass}$ involves $n \log_2 n$ invocations of w to achieve strong mixing of all input blocks (of G) into every output block (of H), so its total computational cost is $O(n \log n)$. The function $\mathsf{reverse\text{-}hourglass}$ is computed in the obvious manner, exploiting the invertibility of w.

Timing. Recall that the butterfly hourglass scheme is a time-based one: It relies on storage access time as its resource bound. As we assume a limit on the server's storage access speed, the effect of the timing bound T in the challenge-response protocol is to impose an upper bound on the amount of data the server can use to compute its response. In our security analysis, we thus rely on the fact that the time bound T translates into an upper bound of at most ϵn storage accesses that the server can make within time T for some $\epsilon < 1$.

We'd like to highlight that we can support any existing storage device (e.g., hard drives, solid-state drives) with this model, but we need to set the parameters ϵ and T depending on the exact characteristics of the storage medium. On rotational drives, for instance, disk access time is highly variable. Sequential block accesses are generally much faster than accesses to blocks at distant locations on disk (or random accesses). Solid-state drives exhibit less variable data access rates. When the challenge-response protocol is run remotely, network latency will also impact the server's response time and needs to be considered when setting the time bound T.

With respect to the above considerations, the use of multiple challenges, i.e., having the client request multiple blocks of H, proves to be a useful approach in smoothing out the variability in storage and network latencies. Below we give a precise analysis for the security of the butterfly construc-

[4]Here we assume n is a power of 2. For n not equal to a power of 2 we may use a different permutation of outputs to inputs between levels. We may also use a higher branching butterfly network.



Algorithm 1: Butterfly hourglass algorithm

1: **for** k from 0 to $n/2^j - 1$ **do**
2: **for** i from 1 to 2^{j-1} **do**
3: $(G_j[i + k \cdot 2^j], G_j[i + k \cdot 2^j + 2^{j-1}]) \leftarrow w(G_{j-1}[i + k \cdot 2^j], G_{j-1}[i + k \cdot 2^j + 2^{j-1}])$;
4: **end for**
5: **end for**

Figure 2: Butterfly hourglass construction: Butterfly graph for $n = 8$ (left) and general algorithm (right).

tion for multiple challenges issued in the challenge-response protocol.

Security analysis. We analyze the butterfly construction using a standard "pebbling" argument that treats w as an atomic operation and intermediate butterfly computations as "pebbles" sitting on the nodes in the butterfly graph. (See, e.g., [9, 10] for similar proofs.) In obtaining a configuration of its storage H', an adversary $\mathcal{A}$ is allowed to play a pebble game. It starts with a given configuration of pebbles in its storage H' that includes both (some of) the original file blocks of F (denoted "red pebbles") and those blocks corresponding to (some of) the intermediary nodes in the butterfly computation from G to H (denoted "black pebbles"). Given a pair of black pebbles in H' input to a w operation in the butterfly graph, they can be replaced in the pebble game with the corresponding output pair of nodes. The goal of the adversary is to find a pebble configuration within a fixed storage bound for H' that will allow him to reply correctly to a large fraction of challenges, and at the same time embed a large number of red pebbles in its storage H'.

Assumptions. Note that this way of modeling an adversary $\mathcal{A}$ through a pebbling game embeds several assumptions about the behavior of the adversary. First, the partitioning assumption defined in Appendix B follows naturally from this model, as the adversary clearly separates its storage H' into red pebbles H'_F (derived from file F) and black pebbles H'_G (derived from formats G and H). Second, the pebbling game encompasses an implicit assumption on w resembling the random oracle model: Output nodes of a w computation in the butterfly graph can only be computed with *complete knowledge* of both input nodes. In this setting, we are able to show a lower bound on the amount of *extra storage* s' that a cheating server needs, defined as the amount of storage needed for answering client challenges besides any plaintext file blocks. (See also Appendix B for a definition of s'.)

THEOREM 1. *Suppose $\mathcal{A}$ can successfully respond to t challenges on randomly selected blocks of size l bits of H with probability α, using extra storage of size s' and up to ϵn timely block accesses. Then the following bound holds:*

$$s' \geq \min\{\alpha^{1/t} nl(1-\epsilon), nl(1-\epsilon) + \log_2 \alpha^{1/t}\} .$$

In other words, the adversary's storage overhead for cheating is linear in its probability of successfully responding to challenges and inversely related to its file-block retrieval rate. Let us give some intuition for the bound in Theorem 1 for $t = 1$ challenges. Ideally, an adversary achieving success probability α of responding to a challenge would need to store $s' = \alpha nl$ bits in its storage (corresponding to all the challenges that the adversary can successfully answer). Nevertheless, there are two strategies for generating additional bits (besides those in its storage) that the adversary can leverage to respond to challenges and improve its success probability. The first strategy is to access a number of at most ϵnl bits in time bound T (and it affects the first term in the bound of Theorem 1). The second strategy is to guess a number of bits in the butterfly network. As the overall probability of success of the adversary in answering challenges is α, an upper bound on the number of bits the adversary can successfully guess is $-\log_2(\alpha)$. These guessed bits affect the second term in the bound of Theorem 1. A detailed proof is given in the full version of the paper.

For example, suppose that the adversary would like to leak plaintext file F through its storage, and it can retrieve file blocks equivalent to at most 5% of F in the allotted response time T. Then to respond successfully to a challenge with 99% probability, $\mathcal{A}$ must incur an 94% overhead, i.e., it must store not just the n blocks of F, but an additional $.94n$ file blocks.

4.2 A permutation-based hourglass function

We now propose an extremely fast and simple hourglass function, one that involves no cryptographic operations. Its security depends upon the performance characteristics of rotational drives. Such drives are optimized for *sequential* file accesses. They perform relatively poorly for *random* accesses, i.e., accesses to widely distributed file locations, as these result in an expensive disk seek operation.

Capitalizing on this characteristic, our hourglass function computes H as a *permutation* of data elements in G. To respond with a challenged block H_i on the fly, then, a cheating server that has only stored partial information about H_i must gather together disparate data elements of G (or F). This process induces a number of random accesses to disk, slowing the server's reply.

Recall that the encoding G of F consists of n blocks of l bits. Let $G_i[j]$ denote the j^{th} *symbol* of G_i, where we define a symbol as a sequence of z bits, e.g., a byte or word, for $z \,|\, l$. Thus $0 \leq j \leq m-1$ with $m = l/z$. Let $G[k]$ denote the k^{th} symbol in the symbolwise representation of G, for $0 \leq k \leq nm - 1$.

Any permutation that scatters the symbols of G widely across H can achieve good security. Our simple hourglass function permutes the symbols of G with the property that for each block G_i the permutation uniformly distributes the

m symbols of G_i over the hourglass blocks in H. That is, each of the hourglass blocks receives $\lceil m/n \rceil$ or $\lfloor m/n \rfloor$ symbols of G_i. In addition, if $m \ll n$, then the permutation composes each block H_i of m symbols from widely staggered positions in G.

We need to emphasize that the block size l of the encryption algorithm needs to be much larger than the symbol size z used for permuting the encrypted file. This will result in a large value of m, the number of symbols within a block. For large m, the intuition behind the security of the permutation-based scheme is as follows. A malicious cloud server that stores a permuted plaintext file has difficulty responding to a challenge consisting of a randomly chosen block $H_i = (G_{i_1}[j_1], \ldots, G_{i_m}[j_m])$ of the hourglass transformation: To compute the k-th encrypted symbol $G_{i_k}[j_k]$ of a challenge block, $1 \leq k \leq m$, the server needs to read a large plaintext block F_{i_v} and encrypt it. Reading the m plaintext blocks $F_{i_1}, \ldots, F_{i_m}$ requires up to m disk seeks. A large value of m results in large disk access time and detection of server misbehavior.

Security analysis. Although the proposed hourglass function is simple, its security analysis is delicate. It relies on a timing model for rotational drives. Rotational drives are designed for optimal access speed over sequential physical locations on a platter. Accesses to blocks at distant location induce a *seek*, a time-consuming physical repositioning of the magnetic head of the drive. In particular, we make the following concrete considerations.

- *Drive model:* For simplicity we assume that a rotational drive has a constant seek time τ_s, i.e., it takes τ_s time to reposition the head. We also assume a constant sequential block-read rate of τ_r. Thus, a sequential read of e blocks is assumed to cost $\tau_s + (e-1)\tau_r$ time.[5]
- *Time bound:* As before, we also consider a particular timing bound T on the response given by an adversarial server $\mathcal{A}$ to a challenge provided by the client.
- *Parallelism:* Of course, a server will in practice stripe files across multiple drives or storage subsystems: Indeed, d-way parallelism enables it to retrieve symbols d times more quickly. This does not, however, actually reduce the extra storage overhead s' that is incurred to the adversary $\mathcal{A}$ by a factor of d. An honest server can seek one block of H faster than a cheating server can gather m symbols across d drives, for $d < m$. In practice, m is on the order of hundreds, and d is much smaller [11], so parallelism has a muted effect on our permutation-based hourglass function. (Note too that if the client can estimate the parallelism d of $\mathcal{A}$, it can request $t = d$ blocks of H during the challenge-response phase.)

Assumptions. Within this model (which assumes a rotational drive model where reading is done at the granularity of blocks), we have analyzed the security guarantees that our permutation-based hourglass function provides. Our analysis relies also on the compression and partitioning assumptions detailed in Appendix B. To reiterate, we assume the adversary cheats by leaking a portion of plaintext file F in its storage. However, the adversary needs to storage additionally s' bits in a string H'_G to help it respond to challenges.

Our full security analysis and accompanying proofs are rather technical and below we state a simplified, but representative, version of our analysis results followed by an example parametrization—in fact, that one we use in our experiments in Section 5.

THEOREM 2. *Let T, τ_s, τ_r, m, l and n be defined as above satisfying $2m \leq l(m - T/\tau_r)$. Suppose $\mathcal{A}$ can successfully respond to a challenge on a randomly selected block of H with probability $\alpha \geq 3/4$, using extra storage of size s'. Then the following bound holds, where $k = \min\{\lfloor \tau_s/\tau_r \rfloor, 1+\lfloor n/(2m^2+4l/3) \rfloor\}$:*

$$s' \geq (2\alpha - 1) \cdot nl \cdot \frac{m - T/(k \cdot \tau_r)}{m - 1} \ .$$

In other words, for large m and α close to 1, the theorem states that s' needs to be at least a fraction $\approx 1 - T/(mk\tau_r)$ of the nl bits that represent F. The intuition behind this is as follows. The quantity $T/(k\tau_r)$ represents an upper bound on the number of file blocks that can be read within time T (either through random seeks or sequential reads). Due to the permutation, each block read (after its encryption) contributes at most one symbol to the reconstruction of a challenge block from H. This means that the additional storage needs to have sufficient information for the reconstruction of the remaining $(1 - T/(mk\tau_r)) \cdot m$ out of m challenge block symbols. For this reason s' needs to be at least a fraction $1 - T/(mk\tau_r)$ of the total number of bits that represent H.

Example parametrization. In our experimental setup, blocks are of size 4KB, as currently used by rotational drives. Since CPU instructions typically operate over 64 bits in today's architectures, we choose a symbol size of $z = 64$ bits. Thus, a block of 4KB consists of $m = 2^9$ symbols. Enterprise drives have an average seek time of around $\tau_s = 6$ms and a disk-to-buffer data transfer rate of ≈ 128KB/ms, hence $\tau_r = 0.03125$ms. Our experiments include a 8GB file, with $n = 2^{21}$ blocks; we use that size for our analysis. We consider a time of $T = 6$ms, that is required to respond to a one-block challenge. (This is much smaller than the network latency between two remote points on the internet. By using multiple challenges we can compensate for network latency.) In this setting, Theorem 2 states that a server $\mathcal{A}$ that leaks the entire plaintext F in its storage and replies correctly to challenges with probability 99% incurs an 89% overhead, i.e., in addition to the n blocks of F $\mathcal{A}$ must store $.89n$ extra blocks.

Implementation in practice. For practical implementations, as in our experiments in section 5, it is reasonable to consider a weak adversary that stores plaintext blocks F_i in their original order. This is the natural behavior of a server, as it enables file segments to be easily retrieved for ordinary use. In this case, the security bounds of our theorem still hold when we implement a highly structured permutation (instead of a uniformly random one) that staggers plaintext symbols at large distances across the hourglass output.

The following choice, used in our experiments, has the

[5] In practice, drive performance is complicated by inexact mapping between logical and physical addresses—due to fragmentation or sector errors—and by non-uniformities in the physical data surface, e.g., the fact that outer rings hold more data than inner ones. But we believe our model is an accurate approximation.

advantages of simplicity and high computational efficiency:

$$H[i] = G[ih \bmod nm] \text{ and } G[i] = H[ig \bmod nm]$$

with $gcd(hg, nm) = 1$.

Another parameter choice in our design is the symbol size z. Smaller values of z offer stronger security, in the sense that the number of blocks m of F whose data make up a block of H scales with $1/z$. CPU instructions, however, operate over data words, typically of 64 bits in today's architectures. So larger values of z (up to 64) mean faster operation for a staggered hourglass function.

4.3 An RSA-based hourglass function

We next present a realization of an hourglass function by applying RSA signing to translate an encoded format G of file F to an hourglass-encapsulated version H of G. At a high level, our protocol employs a hard-to-invert function f in a straightforward way: Each block G_i of the encoded file is mapped to an hourglass block H_i by *inverting G_i under f*. Critical to this protocol, however, is the requirement that f is a *trapdoor* one way permutation, where knowledge of the trapdoor *by the client only, and not by the server,* allows efficient inversion of f. The approach is similar to the example hourglass function described in Section 1, which made use of a compressed-range hash function, but with two important differences. First, for the server, inverting RSA is *computationally infeasible*, *not just moderately difficult to perform* as compressed-range hash functions (or similar puzzles) are. Second, for the client, the transformation from G to H is *easy to perform* using the secret RSA exponent to sign each G_i.

Unlike our previous time-based constructions, here our resource bound is now solely *computation*, and *no assumptions are made on storage usage*. We assume that an adversary $\mathcal{A}$ can do at most l^W work for some $W > 0$, i.e., $\mathcal{A}$ runs in polynomial time in the security parameter defined as the length l of the blocks G_i and H_i (for this construction $|L| = |D| = l$). Thus, based on the cryptographic assumption that inverting the RSA function f is infeasible, translating G_i (or F_i) to H_i without knowledge of the trapdoor is also infeasible for any such $\mathcal{A}$.

Hourglass protocol. If $f : \{0,1\}^l \to \{0,1\}^l$ is a one way trapdoor permutation with trapdoor κ (that is, $f^{-1}(\cdot)$ is hard but $f^{-1}(\kappa, \cdot)$ is easy to compute), then an *f-inversion hourglass protocol* is as follows:

1. The client generates f and κ, and then applies function $\mathsf{hourglass}(f, \kappa, G)$ by individually inverting blocks G_i to $H_i = f^{-1}(G_i)$. The client discards the key sk and sends blocks H_i to the server.
2. On challenge a random index $i \in \{1, \ldots n\}$, the server returns the corresponding inverse image H_i.
3. The client then checks the correctness of the returned answer $\hat{H}_i$ (which might differ from H_i) by running $\mathsf{verify}(H, i, \hat{H}_i)$, and accepts if and only if $H_i = \hat{H}_i$. (Recall that, in practice and as we discussed in Section 3, the authentic format H is available to the client implicitly, through the use of appropriate integrity checks for inverted blocks H_i.)

Note that in the above protocol f, i.e., the RSA signature verification, is used *only to recover* blocks G_i from H_i, in particular, the verification of the challenge blocks $\hat{H}_i$ is done *independently, without using f*.

We instantiate the above protocol using the RSA function $f(x) = x^d \mod N$, its inverse $f^{-1}(y) = y^e \mod N$ and $\kappa = (d, \phi(N))$, where $e > |N|$. Here, l is equal to the modulus size $|N|$. Also, the underlying security parameter ℓ, determined by the best known algorithm for inverting RSA without knowing κ, is polynomially related to l.

Security analysis. To get an intuition of why the above scheme is secure, we note that a computationally bounded $\mathcal{A}$ can't itself feasibly invert blocks under f: To reply to challenge i, it must actually store the original inverted block H_i. Thus, under our partitioning assumption (see Appendix B for more details), to respond correctly to challenges with probability α, $\mathcal{A}$ must store $\approx \alpha n$ blocks in H'_G, i.e., $s' \approx \alpha n l$, yielding an extra storage overhead of factor $\approx \alpha$. Note that as we assume blocks in G are random, $\mathcal{A}$ can't save space by using the same H_i for duplicate blocks of G. (Indeed, when G is uniformly distributed, blocks H_i are statistically independent of each other, and therefore H'_G can be optimally partitioned into $\approx \alpha n$ parts, each containing information about a unique challenge H_i.)

That said, $\mathcal{A}$ *can* "compress" a stored block $H_i = f^{-1}(G_i)$ slightly by throwing away a small number—$O(\log l)$—of bits. By verifying conjectured H_i values against f, it can recompute discarded bits by brute force (with work $O(l)$) when challenged to produce H_i. We refer to as *near-incompressible* any f^{-1} that doesn't admit feasible "compression" of more than the $O(\log l)$ bits of this trivial approach. It is commonly conjectured that RSA signatures with $e > |N|$ are near-incompressible.

We can eliminate this $e > |N|$ conjecture and still get near-incompressible RSA-based signatures by *chaining* applications of f^{-1}. In particular, let

$$g_t^{-1} = (\Pi \circ f^{-1})^t \in \{0,1\}^l \to \{0,1\}^l$$

be a composite inversion function that alternates applications of f^{-1} with applications of a pseudorandom permutation (PRP) Π in a chain of length t, and let us assume that Π is an ideal cipher. If f is a trapdoor permutation that cannot be inverted by a probabilistic algorithm with l^{2W} work with success probability at least ϵ, then for

$$t \geq \lceil (l - W \log l)/(W \log l - \log \epsilon) \rceil$$

the length-t chained inversion function g_t^{-1} can be shown to be near-incompressible.

To get an intuition of why this is true, first note that because f is hard to invert there is a lower bound on the amount by which $H_i = f^{-1}(G_i)$ can be compressed. Also, based on our assumption that Π is an ideal cipher, each of the t blocks that are inverted in the chain is an independent random block, therefore the storage dedicated for challenge block H_i must be further partitioned by the adversary into at least t parts, each storing (a possibly compressed) independent inverted block in the chain. But when the length t of the chain is sufficiently long, the adversary must utilize a relatively large amount of storage for exclusively responding to a challenge H_i, thus making H_i near-incompressible.

Assumptions. Overall, we employ the following assumptions in analyzing the security of the above two schemes, namely the unchained f-inversion protocol and the chained g_t-inversion protocol: (1) The partition assumption (of Appendix B) holds for the adversary $\mathcal{A}$, i.e., its used storage H' can be split into H'_F (to leak raw data) and H'_G (to answer to challenges); (2) The encoded blocks G_i are random,

or equivalently encoding G is uniformly distributed; and (3) either RSA signatures are near-incompressible when $e > |N|$ for an unchained inversion, or the used PRP Π serves as an ideal cipher and sufficiently long chains (as specified above) are used for chained inversions.

Then, for the resulting chained $g = g_t$, with t lower-bounded as above, or the unchained $g = f$, under the $e > |N|$ conjecture, we can prove the following result.

THEOREM 3. *Let f be a trapdoor permutation that cannot be inverted by any algorithm with l^{2W} work with success probability at least ϵ, let $\xi = 2^{-(l-W\log l)}$, and let g be as above. Suppose that $\mathcal{A}$ can successfully respond to challenges in a g-inversion hourglass protocol with probability α, using extra storage of size s' and running in time l^W. Then the following bound holds:*

$$s' \geq \frac{\alpha - \xi}{1 - \xi} \cdot n \cdot (l - W \log l) \ .$$

In other words, under the RSA near-incompressibility assumption or for sufficiently long inversion chains, the (trapdoor permutation) inversion-based hourglass protocol incurs an extra storage overhead to the adversary that is close to the optimal αnl minus the logarithmic-factor savings that the adversary is able to achieve through the trivial compression (via brute force guessing of logarithmic many missing bits) of inverted blocks. In the theorem, the intuition behind ξ is that it is related to[6] the probability that an adversary $\mathcal{A}$ running in time l^W is able to successfully reconstruct/decompress a block of H from a compressed/truncated version of the block of size *less* than $l - W \log l$ bits. This means that in the best case, the adversary needs at least $(l - W \log l)$ bits of storage for $u' = [(\alpha - \xi)/(1 - \xi)]n$ blocks of H such that a total of $u' \cdot 1 + (n - u') \cdot \xi = \alpha n$ blocks of H can be reconstructed. This gives the lower bound on the extra storage s'.

As an example application, consider an RSA key length of $l = 1024 = 2^{10}$ and security of $\ell = 80$ bits (as determined, e.g., by NIST, Pub. 800-57), work of $l^{2W} = 2^{60}$ operations (i.e., $W = 3$), implies a maximum success probability of $\epsilon = 2^{-20}$, and Theorem 3 suggests use of $t = \lceil (l - W \log l)/(W \log l - \log \epsilon) \rceil = \lceil (1024 - 3 \cdot 10)/(3 \cdot 10 + 20) \rceil = 20$ iterations. For $l = 2048$ bits with corresponding security of 112 bits, we similarly get $t = 25$ rounds for the same amount of work ($W = 1.5$, $\epsilon = 2^{-52}$) or $t = 28$ rounds for work of 2^{80} ($W = 2$, $\epsilon = 2^{-32}$). In all cases, the bound on s' is very close to the optimal nl for α close to 1.

5. EXPERIMENTS

We have performed an experimental evaluation in Amazon EC2 of our butterfly construction and one of our permutation constructions. We did not experiment with the RSA construction because the performance of RSA signing operations is well known, thus an evaluation wouldn't provide additional insights. We also acknowledge that the RSA-based construction is less efficient than our other two schemes (in terms of the cost of the hourglass transformation and its inverse), and it's mainly of theoretical interest.

[6]But it is not exactly equal to: In our proof we show that for an adversary running in time l^W the probability that a block from H can be reconstructed from a compressed version of c bits is at most $2^{-(l-c-W\log l)}$. We show that the best strategy for the adversary is to use a compression of $l - W \log l$ bits or of 0 bits for each block.

Table 1: Performance (in seconds) and cost (in cents) of transformation from F to H for file encryption.

File size	Butterfly		Permutation	
	Time	Cost	Time	Cost
1 GB	50.91	0.96	1.64	0.03
2 GB	103.96	1.96	3.24	0.06
4 GB	213.07	4.02	6.45	0.12
8 GB	432.91	8.17	12.73	0.24

We consider a scenario in which the cloud provider (server) runs on Amazon, but isn't Amazon itself. Thus it's subject to Amazon pricing—a fact useful for economic analysis. The tenant acting as a client in our experiments also runs within Amazon EC2.

We implement the butterfly construction using AES encryption with 128-bit file blocks. We parallelize the implementation once the file size exceeds 8MB. For the permutation construction, we consider 4KB file block sizes, and 64-bit symbol sizes as discussed in Section 4.2; as the machine word size, it's a good choice for efficiency. The number of symbols in a block is $m = 512$. We permute the file using the simple construction in Section 4.2. The hourglass file H consists of 512 segments; we include in segment i symbol i of every file block. (Again, this scheme disperses file symbols widely across H.)

We run experiments on Amazon EC2 using a quadruple-extra-large high-memory instance and EBS storage. We also run them on a local machine (i7 980X processor with 6 cores running at 4 GHz). For all our experiments, we show averages over 5 runs.

Hourglass function performance. We first measure the in-memory hourglass function computation time for both the butterfly and permutation constructions for different file sizes. We report computation times in the local machine's memory and in memory for our Amazon EC2 instance in Figure 3.

The butterfly function is at least a factor of 4 faster on our local machine due to hardware support for AES. The butterfly construction also benefits from multiple cores: For large file sizes the multi-threaded implementation is faster by a factor of 5 (on both local and Amazon machines). The permutation scheme permutes machine-size words in main memory, and its cost is determined by cache misses and main memory latency. The implementation on the local machine is about twice as fast as that on Amazon.

Compared to the multi-threaded butterfly implementation, the permutation hourglass function is about 8 times faster on Amazon and 4 times faster locally (since it does not use cryptographic operations). More importantly, this permutation-based hourglass function can be computed in a streaming fashion: After a file block is read from EBS (and before the next block is received), its symbols can be arranged in the corresponding positions in the hourglass output. We performed a preliminary experiment to determine the overhead of such a streamed implementation, in the sense of marginal time above that of simply uploading the file from EBS into main memory. We find that this overhead is negligible. The only extra time the permutation scheme imposes is that of decryption/encryption once the file is uploaded.

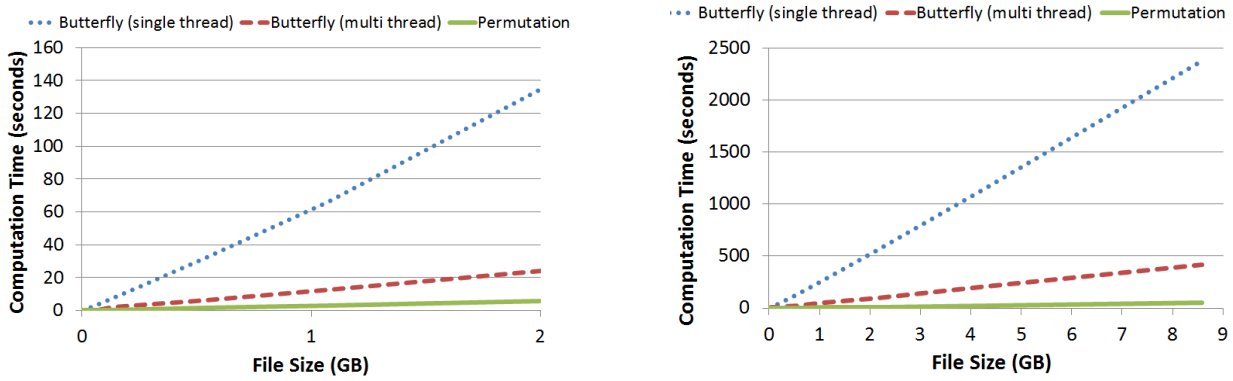


Figure 3: In-memory hourglass function performance for the butterfly and permutation schemes; local machine (left) vs. Amazon EC2 (right).

Table 2: Challenge-response performance

File size	No challenges	Honest	Adversarial
2 GB	2	0.047	8.403
	4	0.062	9.65
	6	0.093	27.40
	8	0.109	26.807
4 GB	2	0.0468	30.321
	4	0.062	101.015
	6	0.078	114.579
	8	0.094	121.431

Economic analysis. We present in Table 1 the total time and cost for computing the transformation from file F to format H for a file-encryption application. (For the permutation scheme, we used the streamed implementation.) We use a cost basis of 68 cents per hour (as charged by EC2 for our instance type).

An honest cloud provider stores transformed file H and computes plaintext F when needed. We described in Section 1 the double-storage problem: A cloud provider might store data in format H to respond correctly in the challenge-response protocol, but *also* store the plaintext for convenience. Using the Amazon EBS pricing scheme for storage (10 cents per GB per month) and the results in Table 1, we argue that this scenario is not economically well motivated. For the butterfly transformation, the cost of computing the plaintext is about 10 times lower than the cost of storing the plaintext (per month). For the permutation scheme, this cost is about 270 times lower than monthly storage. This demonstrates that the butterfly scheme might be used in archival settings (where plaintext data is rarely accessed). On the other hand, the permutation scheme provides economical motivation for the provider to comply and store H even when plaintext accesses are frequent (several hundred times a month).

Challenge-response protocol. We also present in Table 2 the challenge-response protocol times for both honest and adversarial servers (storing plaintext F, but not H). For 2 random challenges, an adversarial server needs to retrieve 1024 symbols distributed across the input file. We observe that the response of an honest server is at least 150 times lower for 2GB files, and at least 650 times lower for 4GB files than that of an adversarial server. We measure the sequential throughput of EBS volumes at around 95MB per second (resulting in 10.77s to read a 1GB file). Based on results in Table 2, once we exceed 6 challenges for 2GB files, (and, respectively 4 challenges for 4GB files), it's faster for an adversarial server to read the full file sequentially rather than access blocks at random. Finally, to demonstrate how the response timing scales, we also plot in Figure 4 the time to read up to 1000 randomly selected blocks from files of different sizes.

The impact of parallelism. An adversary could try to reduce its response time to challenges by spreading file F across multiple EBS volumes in Amazon. Such file distribution improves I/O performance, seek-time latencies, and throughput. In principle, striping a file across v volumes can yield a v-fold performance improvement. Bandwidth constraints suggest that such a strategy would be of limited utility against the butterfly hourglass function, which requires adversarial access to all (or at least a large portion) of F to compute responses. Preliminary experiments suggest that the network interface for a compute instance supports a maximum bandwidth of about 1Gbit/s $\approx$ 125 MB/s. Thus, retrieving a 1GB file, for instance, would require about 8s.[7] By contrast, for an honest service that stores H to respond to a very large challenge, e.g., of 100 blocks, requires less than 1s. (See Figure 4.) Distribution across EBS volumes would be more effective against our permutation hourglass function, where an adversary is constrained by seek-time latency, rather than bandwidth. Still, to achieve response times comparable to those of an honest service, an adversary must distribute F across roughly $v = m$ independent storage systems, i.e., achieve an m-fold seek-time speedup, where m is the number of symbols per challenge block. Our experiments run with $m = 512$ (64-bit symbols and 4096-byte blocks). Experiments suggest, however, that Amazon supports a mere 8-fold speedup in seek times through distribution across EBS volumes [11].

6. RELATED WORK

Hourglass schemes—particularly our proposals based on storage-access speed—intersect technically and conceptually with a few different lines of research.

Economic incentives. Several works have studied the prob-

[7]An adversary would also incur substantial overhead in terms of time and compute-instance cost.

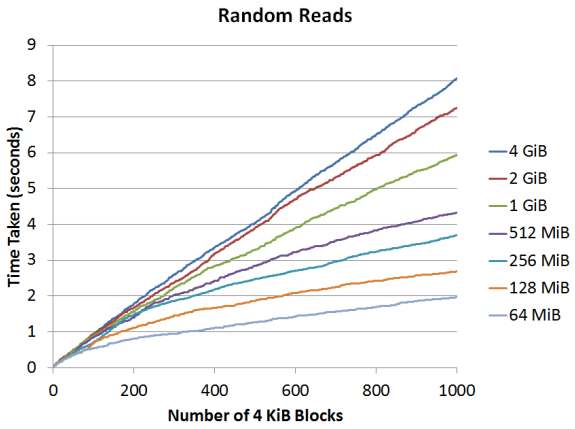


Figure 4: Time to read random blocks from EBS.

lem of creating economic incentives for coordinated protocol participation in various scenarios [22], including peer-to-peer systems, communication networks and information security, examined primarily in the context of algorithmic game theory. Additionally, a growing body of work studies economic incentives and security analysis against a rational adversary in the context of rational cryptography (e.g., [15, 22]) and network security (e.g., [12, 21]).

Puzzles. A moderately hard computational problem is often called a *puzzle* or "proof of work" [17]. Dwork and Naor [8] introduced the idea in a seminal work proposing puzzles as a form of postage to combat spam. Puzzles have also found application in the construction of digital time capsules [25] and denial-of-service resistance [18].

Our storage-bounded hourglass schemes are loosely like puzzles, but rely on storage access as the bounding resource, rather than computation. (Our RSA-based hourglass scheme isn't really puzzle-like.) Additionally, unlike puzzles, hourglass schemes *only* impose high resource requirements on a *cheating* server when it attempts to respond to a challenge, as in, e.g., [28].

Memory-bound functions. Abadi et al. [3] and Dwork, Naor, and Wee [9] explore puzzle variants that depend not on a computational bound, but a bound s on available memory.

In a similar vein, Dziembowski, Kazana, and Wichs [10] introduce the notion of one-time computable pseudorandom functions (PRF). They consider a model in which computation of a PRF $F_K(\cdot)$ requires so much memory that it forces overwriting of the key K itself. Gratzer and Naccache [14] and subsequently Perito and Tsudik [23] similarly propose the idea of "squeezing out" data, namely purging the entire memory of a target device by writing in a long (pseudo)random string and reading it out again.

An hourglass scheme adopts the conceptually related approach of having a server prove correct behavior to a client by showing that it fills its storage with a (nearly) valid encoded version H' (e.g., ciphertext) of the raw file F (e.g., plaintext). The presence of this validly encoded H' in storage "squeezes out" invalidly encoded data (e.g., plaintext) when the associated storage s is bounded.

Storage-enforcing schemes. Building on the notion of "incompressible functions" [7], Golle et al. [13] have proposed schemes in which a server demonstrates that it has devoted a certain, minimum amount of storage to a committed file F. Their protocols, however, don't provide direct assurance that the server has actually stored F itself. Later schemes, including [6, 19, 27] and [5], a variant of [4], enable a server to prove that it has stored some representation of a file F such that F itself can be extracted from the server. But these techniques don't prove anything about the actual representation of F at rest, e.g., that it's encrypted.

Remote posture verification. The Pioneer system [26], and a later variant for mobile handsets [16], remotely times the execution of a software module to measure its trustworthiness. Misconfiguration or the presence of malware slows this execution, creating a measurable delay. Our storage-based hourglass schemes may be viewed as a time-based measurement of a server's posture—not its software posture, but its file-system posture.

7. CONCLUSION

We have introduced hourglass schemes, a new cryptographic construct that enables clients to verify remotely that a server stores files in a particular target format. The formats we have considered here are encryption, encoding with "provenance tags" to enable tracing of leaked files, and the binding together of two (or more) files. Hourglass schemes leverage server resource bounds to achieve their security assurances. We have proposed three main hourglass constructions here. Two draw on storage-access times as a resource bound, another on hardness assumptions on the RSA cryptosystem.

Hourglass schemes hold particular promise as means of monitoring the file-handling practices of cloud services. With this in mind, we have presented a series of experiments demonstrating the feasibility of our proposals in Amazon's cloud service. More generally, as cloud computing proliferates, we believe that hourglass schemes and related techniques will prove a valuable way of penetrating the cloud's abstraction layers and restoring security assurances sacrificed to cloud-based outsourcing.

Acknowledgements

We thank all anonymous reviewers for providing detailed comments and suggestions. Emil Stefanov was supported by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797, a DoD National Defense Science and Engineering Graduate Fellowship and a grant from the Amazon Web Services in Education program.

8. REFERENCES

[1] American Express may have failed to encrypt data. Available at `http://www.scmagazine.com/american-express-may-have-failed-to-encrypt-data/article/170997/`.

[2] Sony playstation data breach, 2011. Available at `http://en.wikipedia.org/wiki/PlayStation_Network_outage`.

[3] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.*, 5:299–327, May 2005.

[4] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.

[5] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT '09*, pages 319–333, Berlin, Heidelberg, 2009.

[6] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, pages 109–127, 2009.
[7] C. Dwork, J. Lotspiech, and M.Naor. Digital signets: self-enforcing protection of digital information. In *STOC*, pages 489–498. ACM, 1996.
[8] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, pages 139–147, 1993.
[9] C. Dwork, M. Naor, and H. Wee. Pebbling and proofs of work. In *CRYPTO*, pages 37–54, 2005.
[10] S. Dziembowski, T. Kazana, and D. Wichs. One-time computable self-erasing functions. In *TCC*, pages 125–143, 2011.
[11] E. Giberti. Honesty box: EBS performance revisited. Blog posting, available at http://tinyurl.com/3nqxngv, 2010.
[12] S. Goldberg, S. Halevi, A. D. Jaggard, V. Ramachandran, and R. N. Wright. Rationality and traffic attraction: incentives for honest path announcements in BGP. In *SIGCOMM*, pages 267–278, 2008.
[13] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *FC '02*, pages 120–135, 2003.
[14] V. Gratzer and D. Naccache. Alien vs. quine. *IEEE Security and Privacy*, 5(2):26–31, 2007.
[15] J. Halpern and V. Teague. Rational secret sharing and multiparty computation: extended abstract. In *STOC*, pages 623–632, 2004.
[16] M. Jakobsson and K. Johansson. Retroactive detection of malware with applications to mobile platforms. In *HotSec*, pages 1–13, 2010.
[17] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, pages 258–272, 1999.
[18] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, pages 151–165, 1999.
[19] A. Juels and B. S. K. Jr. PORs: proofs of retrievability for large files. In *ACM CCS*, pages 584–597, 2007.
[20] M. Labs and M. F. P. Services. Protecting your critical assets: Lessons learned from "Operation Aurora", 2010. Whitepaper available at `http://www.mcafee.com/us/resources/white-papers/wp-protecting-critical-assets.pdf`.
[21] M. H. Manshaei, Q. Zhu, T. Alpcan, and J.-p. Hubaux. Game theory meets network security and privacy. *Main*, V(April):1–44, 2010.
[22] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.
[23] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, pages 643–662, 2010.
[24] R. Rivest. All-or-nothing encryption and the package transform. In *Fast Software Encryption*, pages 210–218, 1997.
[25] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, 1996.
[26] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP*, pages 1–16, 2005.
[27] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, 2008.
[28] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, pages 601–612, 2005.

APPENDIX

A. ENCODINGS

We now elaborate on Phase 1 of our generic hourglass protocol that we presented in Section 3. We give more details on how the server generates an encoded version of F, as well as a proof of correct encoding. In both the case of encryption and watermarking encoding, the provider must encode file F into G using a secret *unknown* to the verifier, i.e., the client. In the case of encryption, the secret is the key κ. For watermarking, the secret consists of multiple digital signatures produced by the server that attest to the provenance of the file; the signatures, if divulged, can be used to frame the provider. Thus proving that G is correctly encoded creates a difficulty: The cloud provider has provided the verifier (client) with the encoded file G and must prove to the verifier that G is *correctly* computed from F, but *without revealing* the secret used to produce encoding G.

Encryption encoding. For the case of encryption, the idea is as follows. The prover encodes F under a pseudorandom permutation (PRP), and partitions the file into n blocks. It uses master key κ to derive a set of keys $\{\kappa_i\}_{i=1}^{n}$, encrypting the i^{th} block of the file under κ_i. To verify that G is correctly formatted, the verifier challenges the prover to reveal a subset of the keys for randomly chosen blocks. The PRP ensures that revealing a subset of the shares does not reveal κ and therefore doesn't permit decryption of the full file.

Figure 5 (left) specifies our protocol for a server to prove to a client that G represents a correct encryption of F. Here, $PRP_{\kappa'}$ denotes a keyed PRP, and E_{κ_i,κ^*} denotes encryption under keys κ_i and κ^* (the two keys could be hashed together, for instance, to obtain the file block encryption key). Also, $KD(\kappa, i)$ denotes an indexed key-derivation function that takes master key κ as input. The number q of challenged blocks can be adjusted to obtain a desired confidence probability that file blocks are correctly encrypted.

The client supplies some randomness (in the form of a key κ^*) that is used in combination with the secret key generated by the server for generating file block encryption keys. The randomness provided by the client serves the goal of enforcing that file block encryption keys are generated with proper randomness.

As with any cryptographic scheme, it is important that the server protect the encryption key κ. There are various well-studied techniques for ensuring that the key is kept secret such as using hardware security modules, trusted hardware, or secure co-processors.

Watermark encoding. Figure 5 (right) specifies our protocol for a server to prove to a client that G represents the application of an hourglass function to F under a correct incorporation of a provenance tag π. We let $\sigma(M)$ denote a digital signature by the server on message M.

The main idea is to divide the file into n blocks and embed a signature σ_i with each block F_i so that no block F_i can be retrieved without revealing the signature embedded with F_i. This property is achieved by applying an AONT transformation [24] to each file block and the corresponding signature. Block G_i of encoded file G is then computed as $\mathsf{AoNT}[\sigma_i, F_i]$, where σ_i is a signature by the server on the file handler $\mathsf{handler}$ and block index i.

The provenance tag π consists of the file handler and hashes on signatures σ_i and is published by the server. A proof of leakage by an external entity consists of a number v of correct signatures, i.e., v signatures corresponding to hashed signatures in π. The value v is a security parameter such that $q < v \leq n$.

The challenge procedure is exactly like that for file encryption. The client challenges q randomly selected segments

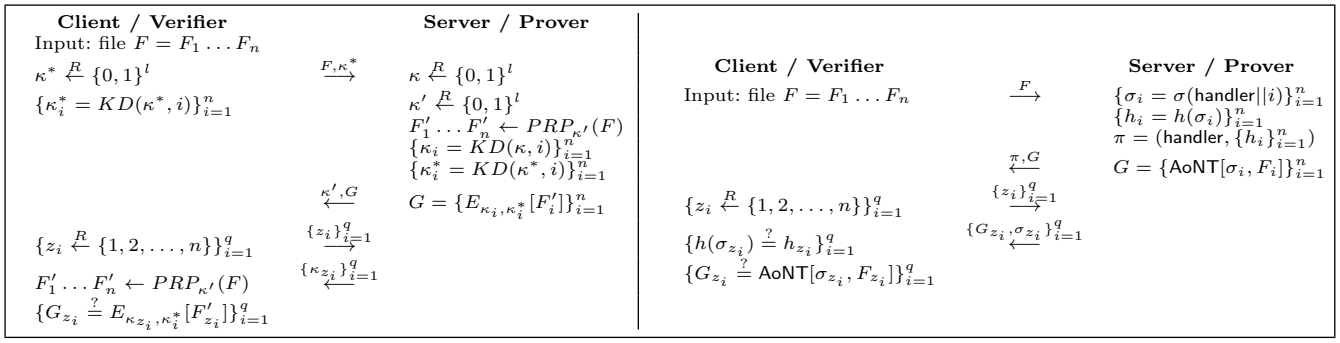


Figure 5: Proof of correct encodings: The cases of file encryption (left) and watermarking encoding (right).

of the file, and the server replies with the corresponding signatures. The client verifies that the AoNT encoding is performed correctly on the challenged segments and that signature hashes match those published in the provenance tag. Using large sized blocks reduces the additional storage expansion for signatures. At the same time, a large block size reduces the challenge space and incurs overhead in the challenge procedure of Phase 1, as large file blocks have to be retrieved to check signature correctness. Thus, we have to achieve a balance among different metrics of interest. A good choice is a block size of $O(\sqrt{|F|})$ bits, resulting in $n = O(\sqrt{|F|})$ blocks.

File binding encoding. In some scenarios, it might be useful to verify that multiple files are stored together (i.e., that they have been bound together). This feature can be useful if, for example, we want to verify that source code is stored together with its corresponding license. The protocol for file binding can very easily be constructed from watermarking. A pair of files, F_1 and F_2 are bound together or encoded via application of AoNT. Subsequent application of an hourglass function, setup, and the challenge-response protocol are then similar to the watermarking encoding protocol. This can easily be generalized to any number of files.

B. FORMAL SECURITY DEFINITION

We assume that the server is controlled by an adversary $\mathcal{A}$ who has resources polynomially bounded in l. We let s denote an upper bound on the size of $\mathcal{A}$'s storage (expressed in bits). Both s and the file size n are polynomial in l. Also, $\mathcal{A}$ is stateless, in the sense that distinct adversarial algorithms (e.g., $\mathcal{A}(\texttt{Store})$ and $\mathcal{A}(\texttt{ChalRes})$) cannot intercommunicate. (The two algorithms represent the state of cloud at different times, so the only channel between them is the stored value H.)

General security definition. Figure 6 presents the experiment $\mathbf{Exp}_{\mathcal{A}}^{HG}[l, n, s, \delta]$ that characterizes the security of general hourglass schemes. We define $succ_{\mathcal{A}}^{HG}[l, n, s, \delta] \triangleq \Pr[\mathbf{Exp}_{\mathcal{A}}^{HG}[l, n, s, \delta] = 1]$.

We found that coming up with the right definition for an hourglass system (one of the paper's technical contribution) is quite subtle. The adversary's objective is to *leak* F, i.e., to store a file representation from which F can be recovered without decoding/decryption. We model the adversary's goal, however, as leakage of a random string ρ embedded in F. We can think of ρ as the underlying entopy of F or a maximally compressed version of F. If F is compressible,

Security experiment $\mathbf{Exp}_{\mathcal{A}}^{HG}[l, n, s, \delta]$:

Initialize:
$\rho \xleftarrow{R} \{0,1\}^{\delta s}$
$F \in B^n \leftarrow \mathcal{A}(\texttt{FileGen}, \rho)$
$\kappa_1 \leftarrow \text{keygen-enc}(l)$
$(\kappa_2, \kappa_3) \leftarrow \text{keygen-hg}(l)$
$G \in L^n \leftarrow \text{encode}(\kappa_1, F)$
$H \in D^n \leftarrow \text{hourglass}(\kappa_2, \kappa_3, G)$
Generate Storage:
$H' \in \{0,1\}^s \leftarrow \mathcal{A}^{\text{encode}(\kappa_1,\cdot),\text{decode}(\kappa_1,\cdot)}(\texttt{Store}, \rho, F, G, H, \kappa_2)$
Recover Raw Data:
$\rho' \leftarrow \mathcal{A}(\texttt{RecRaw}, H', \kappa_2)$
Challenge-Response:
$c \xleftarrow{R} \text{challenge}$
$r \leftarrow \mathcal{A}^{\text{storage}(H',\cdot),\text{encode}(\kappa_1,\cdot),\text{decode}(\kappa_1,\cdot)}(\texttt{ChalRes}, c, \kappa_2)$
Finalize:
return $(\rho' \stackrel{?}{=} \rho)$ AND $(\text{verify}(H, c, r) \stackrel{?}{=} 1)$

Figure 6: General hourglass security experiment.

then it is easier for the adversary to recover ρ than to recover the longer representation F directly. In our experiment, the adversary may encode ρ in F arbitrarily.

In an honest execution of the protocol, a server will store a file encoding H that has two properties: (1) By accessing stored data format H, the server can respond correctly to client challenges with overwhelming probability; and (2) H is a function of G and, in this sense, prevents disclosure of ρ unless decode is called.

In contrast, the goal of the adversary $\mathcal{A}$ is to construct some H' for storage with a different pair of properties, namely: (1) By accessing stored data H', the adversary can respond correctly, with high probability, to client challenges; and (2) The adversary can extract ρ from H' without calling decode. In other words, $\mathcal{A}$ would like to appear to store a valid file encoding H that does not contain raw data, but actually store a file encoding H' that leaks raw data.

In the experiment above, we model storage as an oracle $\text{storage}(H', \cdot)$ that reflects resource bounds on access to stored file $H' \in L^{s/l}$ of size s bits. Oracle $\text{storage}(H', \cdot)$ takes an index i as input, and outputs either the i-th block of H', or $\perp$ if the modeled resource bound has been exceeded.

On modeling leakage. In our experiment $\mathcal{A}$ aims to recover ρ from H' (while correctly answering challenges). Our experiment requires that $\mathcal{A}(\texttt{RecRaw}, \cdot)$ do so *without access to* $\text{decode}(\kappa_1, \cdot)$. The decode oracle models the underlying cryptographic access-control mechanism. For instance, in practice, decode might correspond to a module based on trusted

hardware that decrypts ciphertext data. Denying adversary $\mathcal{A}(\texttt{RecRaw}, \cdot)$ access to decode models the presumption that an attacker can't breach the trusted hardware module. (For technical reasons, it also doesn't have access to $\mathsf{encode}(\kappa_1, \cdot)$.)

Compression assumption. We assume that the output of the encode function G on input raw file $F \in B^n$ is uniformly distributed in domain L^n. G is thus a string independent of plaintext file F that can not be further compressed by the adversary. This assumption is necessary in our security analysis in order to provide meaningful lower bounds on the amount of extra storage incurred by a cheating server.

We briefly justify the compression assumption for our three encodings of interest detailed in Appendix A. For the encryption encoding, we can model the $(\mathsf{encode}, \mathsf{decode})$ operations as an ideal cipher. In this case, the ciphertext G resulting after encrypting plaintext file F is uniformly distributed and independent on the plaintext. For the watermark encoding, the all-or-nothing transformation AoNT can be modeled as a random oracle, resulting again in uniformly distributed output G. File binding encoding is a simple extension of watermark encoding and as such the same random oracle modeling of the AoNT transform can be used to justify the compression assumption.

Partitioning assumption. We make a simplifying technical assumption on $\mathcal{A}$ throughout our security proofs (which are omitted for lack of space). This *partitioning assumption* requires that $\mathcal{A}(\texttt{Store}, \cdot)$ output H' of the form $H' = (H'_F \parallel H'_G)$, where H'_F and H'_G are distinct substrings as follows:

- H'_F is a raw representation of F. In other words, $\mathcal{A}$ can recover ρ from H'_F, i.e., there is an algorithm $\mathcal{A}(\texttt{RecRaw}_{\mathtt{H}'_\mathtt{F}}, H'_F, \kappa_2) \rightarrow \rho$.
- H'_G is "extra" storage used by $\mathcal{A}$ to help it respond to challenges. We can think of H'_G as the storage overhead imposed on $\mathcal{A}$ in order to cheat and answer challenges correctly while it is leaking F. We assume that $\mathcal{A}$ computes H'_G over G and H, without knowledge of F. (But $\mathcal{A}$ is allowed to use all of H' to respond to challenges.)

The partitioning assumption renders security analysis of our concrete hourglass constructions simpler. We conjecture, however, that the assumption doesn't in fact result in a weaker class of adversary.[8] It seems implausible that *mixing together* H'_F and H'_G into a combined representation would advantage $\mathcal{A}$. After all, G is uniformly distributed in domain L^n and therefore an independent string in the view of $\mathcal{A}$. We leave this conjecture on the partitioning assumption as an open problem. In our security analysis, we use s' to denote $|H'_G|$.

[8]In particular, our conjecture applies to hourglass schemes that are *valid*, as defined in Section 2.2. Clearly, for a degenerate scheme with challenges constructed over F, $\mathcal{A}$ would benefit from computing H'_G as a function of F.