HAIL: A High-Availability and Integrity Layer for Cloud Storage

Kevin D. Bowers
RSA Laboratories
Cambridge, MA, USA
kbowers@rsa.com

Ari Juels
RSA Laboratories
Cambridge, MA, USA
ajuels@rsa.com

Alina Oprea
RSA Laboratories
Cambridge, MA, USA
aoprea@rsa.com

ABSTRACT

We introduce HAIL (High-Availability and Integrity Layer), a distributed cryptographic system that allows a set of servers to prove to a client that a stored file is intact and retrievable. HAIL strengthens, formally unifies, and streamlines distinct approaches from the cryptographic and distributed-systems communities. Proofs in HAIL are efficiently computable by servers and highly compact—typically tens or hundreds of bytes, irrespective of file size. HAIL cryptographically verifies and reactively reallocates file shares. It is robust against an active, mobile adversary, i.e., one that may progressively corrupt the full set of servers. We propose a strong, formal adversarial model for HAIL, and rigorous analysis and parameter choices. We show how HAIL improves on the security and efficiency of existing tools, like Proofs of Retrievability (PORs) deployed on individual servers. We also report on a prototype implementation.

Categories and Subject Descriptors

E.3 [**Data**]: [Data Encryption]

General Terms

Security

Keywords

Distributed storage systems, cloud storage, data availability, erasure codes, proofs of retrievability

1. INTRODUCTION

Cloud storage denotes a family of increasingly popular on-line services for archiving, backup, and even primary storage of files. Amazon S3 [1] is a well known example. Cloud-storage providers offer users clean and simple file-system interfaces, abstracting away the complexities of direct hardware management. At the same time, though, such services eliminate the direct oversight of component reliability and security that enterprises and other users with high service-level requirements have traditionally expected.

To restore security assurances eroded by cloud environments, researchers have proposed two basic approaches to client verification of file availability and integrity. The cryptographic community has proposed tools called proofs of retrievability (PORs) [23] and proofs of data possession (PDPs) [2]. A POR is a challenge-response protocol that enables a prover (cloud-storage provider) to demonstrate to a verifier (client) that a file F is retrievable, i.e., recoverable without any loss or corruption. The benefit of a POR over simple transmission of F is efficiency. The response can be highly compact (tens of bytes), and the verifier can complete the proof using a small fraction of F.

As a standalone tool for testing file retrievability against a single server, though, a POR is of limited value. Detecting that a file is corrupted is not helpful if the file is irretrievable and the client has no recourse. Thus PORs are mainly useful in environments where F is distributed across multiple systems, such as independent storage services. In such environments, F is stored in redundant form across multiple servers. A verifier (user) can test the availability of F on individual servers via a POR. If it detects corruption within a given server, it can appeal to the other servers for file recovery. To the best of our knowledge, the application of PORs to distributed systems has remained unexplored in the literature.

A POR uses file redundancy *within a server* for verification. In a second, complementary approach, researchers have proposed distributed protocols that rely on queries *across servers* to check file availability [25, 31]. In a distributed file system, a file F is typically spread across servers with redundancy—often via an erasure code. Such redundancy supports file recovery in the face of server errors or failures. It can also enable a verifier (e.g., a client) to check the integrity of F by retrieving fragments of F from individual servers and cross-checking their consistency.

In this paper, we explore a unification of the two approaches to remote file-integrity assurance in a system that we call *HAIL (High-Availability and Integrity Layer)*.

HAIL manages file integrity and availability across a collection of servers or independent storage services. It makes use of PORs as building blocks by which storage resources can be tested and reallocated when failures are detected. HAIL does so in a way that transcends the basic single-server design of PORs and instead exploits both within-server redundancy and cross-server redundancy.

HAIL relies on a single trusted verifier—e.g., a client or a service acting on behalf of a client—that interacts with servers to verify the integrity of stored files. (We do not consider a clientless model in which servers perform mutual verification, as in distributed information dispersal algorithms such as [16, 17, 8, 20].)

HAIL offers the following benefits:

Strong file-intactness assurance: HAIL enables a set of servers to prove to a client via a challenge-response protocol that a stored file F is fully intact—more precisely, that the client can recover F with overwhelming probability. HAIL protects against even small, e.g., single-bit, changes to F.

Low overhead: The per-server computation and bandwidth required for HAIL is comparable to that of previously proposed PORs. Apart from its use of a natural file sharing across servers, HAIL improves on PORs by eliminating check values and reducing within-server file expansion.

Strong adversarial model: HAIL protects against an adversary that is *active*, i.e., can corrupt servers and alter file blocks and *mobile*, i.e., can corrupt every server over time.

Direct client-server communication: HAIL involves one-to-one communication between a client and servers. Servers need not intercommunicate—or even be aware of other servers' existence. (In comparison, some information dispersal algorithms involve server-to-server protocols, e.g., [16, 17, 8, 20].) The client stores just a secret key.

Static file protection: HAIL protects static stored objects, such as backup files and archives. Constructing protocols to accommodate file updates, i.e., to provide integrity assurance for dynamically changing objects, is left to future work.

Our two broad conceptual contributions in HAIL are:

Security modeling. We propose a strong, formal model that involves a *mobile adversary*, much like the model that motivates proactive cryptographic systems [22, 21]. A mobile adversary is one capable of progressively attacking storage providers—and in principle, ultimately corrupting all providers at different times.

None of the existing approaches to client-based file-integrity verification treats the case of a mobile adversary. We argue that the omission of mobile adversaries in previous work is a serious oversight. In fact, we claim that *a mobile adversarial model is the only one in which dynamic, client-based verification of file integrity makes sense*. The most common alternative model is one in which an adversary (static or adaptive) corrupts a bounded number of servers. As real-world security model for long-term file storage, this approach is unduly optimistic: It assumes that some servers are *never* corrupted. More importantly, though, an adversarial model that assumes a fixed set of honest servers for all time does not require dynamic integrity checking at all: A robust file encoding can guarantee file recovery irrespective of whether or not file corruptions are detected beforehand.

HAIL design strategy: Test-and-Redistribute (TAR). HAIL is designed like a proactive cryptographic system to withstand a mobile adversary. But HAIL aims to protect integrity, rather than secrecy. It can therefore be *reactive*, rather than proactive. We base HAIL on a new protocol-design strategy that we call TAR (Test-And-Redistribute). With TAR, the client uses PORs to detect file corruption and trigger reallocation of resources when needed—and only when needed. On detecting a fault in a given server via challenge-response, the client communicates with the other servers, recovers the corrupted shares from cross-server redundancy built in the encoded file, and resets the faulty server with a correct share.

Our TAR strategy reveals that for many practical applications, PORs and PDPs are *overengineered*. PORs and PDPs assume a need to store explicit check values with the prover. In a distributed setting like that for HAIL, it is possible to obtain such check values from the collection of service providers itself. On the other hand, distributed protocols for checking file availability are largely *under-engineered*: Lacking robust testing and reallocation, they provide inadequate protection against mobile adversaries.

Three main coding constructions lie at the heart of HAIL:

Dispersal code: In HAIL, we use what we call a *dispersal code* for robustly spreading file blocks *across servers*. For the dispersal code in HAIL, we propose a new cryptographic primitive that we call an *integrity-protected error-correcting code* (IP-ECC). Our IP-ECC construction draws together PRFs, ECCs, and universal hash functions (UHFs) into a single primitive. This primitive is an error-correcting code that is, at the same time, a corruption-resilient MAC on the underlying message. The additional storage overhead is minimal—basically just one extra codeword symbol.

In a nutshell, our IP-ECC is based on three properties of (certain) universal hash function families h: (1) h is linear, i.e., $h_\kappa(m) + h_\kappa(m') = h_\kappa(m+m')$ for messages m and m' and key κ; (2) For a pseudorandom function family (PRF) g, the function $h_\kappa(m) + g_{\kappa'}(m)$ is a cryptographic message-authentication code (MAC) on m; and (3) $h_\kappa(m)$ may be treated as a parity block in an error-correcting code applied to m.

Server code: File blocks *within* each server are additionally encoded with an error-correcting code that we call a *server code*. This code protects against the low-level corruption of file blocks that may occur when integrity checks fail. (For efficiency, our server code is a computational or "adversarial" error-correcting code as defined in Bowers et al. [6].)

Aggregation code: HAIL uses what we call an *aggregation code* to compress responses from servers when challenged by the client. It acts across multiple codewords of the dispersal code. One feature of the aggregation code is that it combines / aggregates multiple MACs in our IP-ECC into a single *composite MAC*. This composite MAC verifies correctly only if it represents a combination of valid MACs on each of the aggregated codewords.

Note that while the aggregation code is built on an error-correcting code, it is computed as needed, and thus imposes no storage or file-encoding overhead.

Organization. We review related work in Section 2. We give an overview of the HAIL construction and its main technical ingredients in Section 3. We introduce our adversarial model in Section 4 and describe technical building blocks for HAIL in Section 5. The details of the HAIL protocol are described in Section 6, and its security properties in Section 7. We give implementation results in Section 8 and conclude in Section 9.

2. RELATED WORK

HAIL may be viewed loosely as a new, service-oriented version of RAID (Redundant Arrays of Inexpensive Disks). While RAID manages sector redundancy dynamically across hard-drives, HAIL manages file redundancy across cloud storage providers. Recent multi-hour failures in S3 illustrate the need to protect against basic service failures in cloud environments. In view of the rich targets for attack that cloud storage providers will present, HAIL is designed to withstand Byzantine adversaries. (RAID is mainly designed for crash-recovery.)

Information dispersal. Distributed information dispersal algorithms (IDA) that tolerate Byzantine servers have been proposed [16, 17, 8, 20]. In these algorithms, file integrity is enforced within the pool of servers itself. Some protocols protect against faulty clients that send inconsistent shares to different servers [17, 8, 20]. In contrast, HAIL places the task of file-integrity checking in the hands of the client or some other trusted, external service. Unlike previous work, which verifies integrity at the level of individual file blocks, HAIL provides assurance at the granularity of a full file. This difference motivates the use of PORs in HAIL, rather than block-level integrity checks.

Universal Hash Functions. Our IP-ECC primitive fuses several threads of research that have emerged independently. At the heart of this research are *Universal Hash-Functions* (UHFs). (In the distributed systems literature, common terms for variants of UHFs are *algebraic signatures* [31] or *homomorphic fingerprinting* [20].) UHFs can be used to construct *message-authentication codes (MAC)* [19, 4, 14] (see [28] for a performance evaluation of various schemes). In particular, a natural combination of UHFs with pseudorandom functions (PRFs) yields MACs.

PORs and PDPs. Juels and Kaliski (JK) [23] formally define PORs and propose a POR protocol that only supports a limited number of challenges. Shacham and Waters (SW) [32] offer alternative constructions based on the idea of storing block integrity values that can be aggregated to reduce the communication complexity of a proof. They construct symmetric-key POR protocols based on the UHF + PRF paradigm and publicly verifiable PORs based on publicly verifiable homomorphic authenticators.

In concurrent and independent work, Bowers et al. [6] and Dodis et al. [12] give frameworks for POR protocols that generalize the JK and SW protocols. Both papers propose the use of an error-correcting code in computing server responses to challenges with the goal of ensuring file extraction through the challenge-response interface. The focus of [12] is mostly theoretical in providing extraction guarantees for adversaries replying correctly to an arbitrary small fraction of challenges. In contrast, Bowers et al. consider POR protocols of practical interest (in which adversaries with high corruption rates are detected quickly) and show different parameter tradeoffs when designing POR protocols.

Ateniese et al. [2] propose a closely related construction called a *proof of data possession* (PDP). A PDP detects a large fraction of file corruption, but does not guarantee file retrievability. Subsequent work gives file update protocols in the PDP model [3, 13]. Curtmola et al. [11] proposed an extension of PDPs to multiple servers. Their proposal essentially involves computational cost reduction through PDP invocations across multiple replicas of a single file, rather than a share-based approach. Earlier closely related constructions to PORs and PDPs include [15, 33, 27].

Distributed protocols for dynamic file-integrity checking. Lillibridge et al. [25] propose a distributed scheme in which blocks of a file F are dispersed across n servers using an (n, m)-erasure code. Servers spot-check the integrity of one another's fragments using message authentication codes (MACs).

Schwartz and Miller (SM) [31] propose a scheme that ensures file integrity through distribution across multiple servers, using error-correcting codes. They employ keyed algebraic encoding and stream-cipher encryption to detect file corruptions. Their keyed encoding function is equivalent to a Reed-Solomon code in which codewords are generated through keyed selection of symbol positions. We adopt some ideas of simultaneous MACing and error-correcting in our HAIL constructions, but we define the construction rigorously and prove its security properties.

Proactive cryptography. Our adversarial model is inspired by the literature on proactive cryptography initiated by [22], which has yielded protocols resilent to mobile adversaries for secret sharing (e.g., [22, 7]) as well as signature schemes (e.g., [21]). In previous proactive systems, key compromise is a silent event; consequently, these systems must redistribute shares *automatically* and provide protections that are *proactive*. Corruption of a stored file, however, is not a silent event. It results in a change in server state that a verifier can detect. For this reason, HAIL can rely on remediation that is *reactive*. It need not automatically refresh file shares at each interval, but only on detecting a fault.

3. HAIL OVERVIEW

In this section, we present the key pieces of intuition behind HAIL. We start with simple constructions and build up to more complex ones.

In HAIL, a client distributes a file F with redundancy across n servers and keeps some small (constant) state locally. The goal of HAIL is to ensure resilience against a *mobile adversary*. This kind of powerful adversary can potentially corrupt all servers across the full system lifetime. There is one important restriction on a mobile adversary, though: It can control only b out of the n servers within any given time step. We refer to a time step in this context as an *epoch*.

In each epoch, the client that owns F (or potentially some other entity on the client's behalf) performs a number of checks to assess the integrity of F in the system. If corruptions are detected on some servers, then F can be reconstituted from redundancy in intact servers and known faulty servers replaced. Such periodic integrity checks and remediation are an essential part of guaranteeing data availability against a mobile adversary: Without integrity checks, the adversary can corrupt all servers in turn across $\lceil n/b \rceil$ epochs and modify or purge F at will.

Let us consider a series of constructions, explaining the shortcomings of each and showing how to improve it. In this way, we introduce the full conceptual complexity of HAIL incrementally.

Replication system.

A first idea for HAIL is to replicate F on each of the n servers. Cross-server redundancy can be used to check integrity. To perform an integrity check, the client simply chooses a random file-block position j and retrieves the corresponding block F_j of F from each server. Provided that all returned blocks are identical, the client concludes that F is intact in that position. If it detects any inconsistencies, then it reconstructs F (using majority decoding across servers) and removes / replaces faulty servers. By sampling multiple file-block positions, the client can boost its probability of detecting corruptions.

A limitation of this approach is that the client can only feasibly inspect a small portion of F. Another is that while the client checks *consistency* across servers, it does not directly check *integrity*, i.e., that the retrieved block for position j is the one originally stored with F. Consequently, this simple approach is vulnerable to a *creeping-corruption attack*. The adversary picks a random position i and changes the original block value F_i to a corrupted value $\hat{F}_i$ in all b servers corrupted during a given epoch. After $T = \lceil n/(2b) \rceil$ epochs, the adversary will have changed F_i to $\hat{F}_i$ on a majority of servers. At that point, majority decoding will fail to reconstruct block F_i.

Because the client can feasibly check only a small fraction of the file, the probability that it will detect temporary inconsistencies introduced by the adversary's corruptions is low. Thus, the adversary can escape detection and render F unretrievable with high probability in T epochs.

Replication system with POR.

To achieve better resilence against a creeping-corruption attack, we might employ a POR system (e.g., [23, 32, 6]) on each of the n servers. In a single-server POR system, F is encoded under an error-correcting code (or erasure code) that we refer to in HAIL as the *server code*. The server code renders each copy of F robust against a fraction ϵ_c of corrupted file blocks, protecting against the single-block corruptions of our previous approach. (Here ϵ_c is the error rate of the server code.)

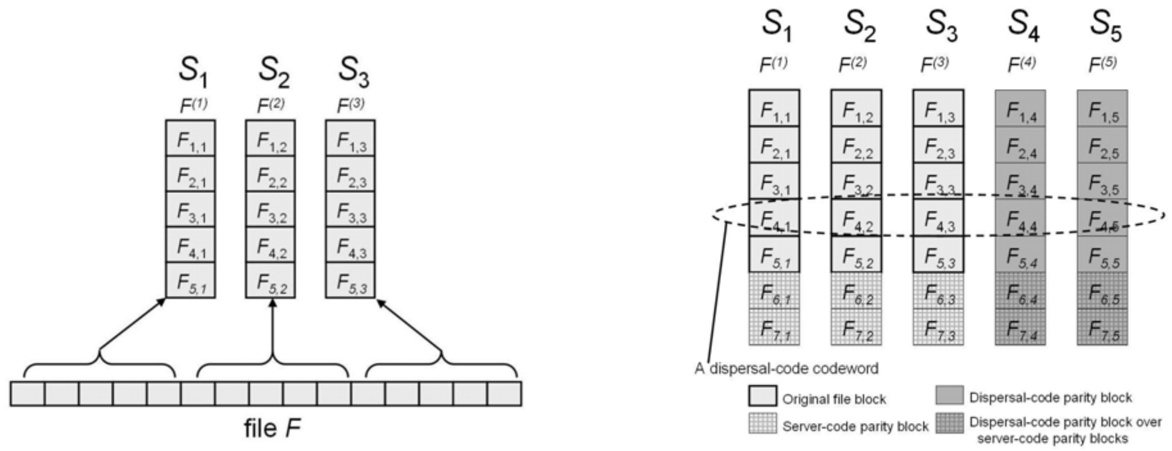

S_1 S_2 S_3
$F^{(1)}$ $F^{(2)}$ $F^{(3)}$
file F
S_1 S_2 S_3 S_4 S_5
$F^{(1)}$ $F^{(2)}$ $F^{(3)}$ $F^{(4)}$ $F^{(5)}$
A dispersal-code codeword
Original file block
Server-code parity block
Dispersal-code parity block
Dispersal-code parity block over server-code parity blocks

Figure 1: Encoding of file F: on the left, original file represented as a matrix; on the right, encoded file with parity blocks added for both the server and dispersal codes.

There are then two options to check the integrity of F. One is to use the single-server POR approach of embedding integrity checks within each server's copy of F. This approach, however, imposes high storage overhead: It does not take advantage of cross-server redundancy.

An alternative approach is to perform integrity checks by comparing block values in a given position j using cross-server redundancy as in our previous construction. With this approach, the system is still vulnerable to a creeping-corruption attack, but much less than in the previous construction. Suppose that the POR can detect inconsistencies within a server if the adversary modifies at least ϵ_d-fraction of blocks. Assuming that the client performs majority decoding to replace faulty servers whenever it detects corruption, this approach will ensure the integrity of F with high probability for $T = \lceil n/(2b) \rceil \times (\epsilon_c/\epsilon_d)$ epochs—improving over the previous approach by a factor of ϵ_c/ϵ_d.

Dispersal code with POR.

We can improve the storage overhead of the previous approach with a more intelligent approach to creating file redundancy across servers. Rather than replicating F across servers, we can instead distribute it using an error-correcting (or erasure) code. We refer to this code in HAIL as the *dispersal code*. In HAIL, each file block is individually distributed across the n servers under the dispersal code.

Let (n, ℓ) be the parameters of the dispersal code. We assume for convenience that this code is systematic, i.e., that it preserves ℓ message blocks in their original form. Then ℓ is the number of *primary servers*, those servers that store fragments of the original file F. The remaining $n - \ell$ are *secondary servers*, or redundant servers, i.e., servers that maintain additional redundancy/parity blocks and help recover from failure.

A graphical representation of dispersal encoding is given in Figure 1. Before transforming the file F into a distributed, encoded representation, we partition it into ℓ distinct segments $F^{(1)}$, $\ldots, F^{(\ell)}$ and distribute these segments across the primary servers $S_1, \ldots, S_\ell$. This distributed cleartext representation of the file remains untouched by our subsequent encoding steps. We then encode each segment $F^{(j)}$ under the server code with error rate ϵ_c. The effect of the server code is to extend the "columns" of the encoded matrix by adding parity blocks. Next, we apply the dispersal code to create the parity blocks that reside on the secondary servers. It extends the "rows" of the encoded matrix across the full set of n servers $S_1, \ldots, S_n$.

With this scheme, it is possible to use cross-server redundancy to check the integrity of F. The client / verifier simply checks that the blocks in a given position, i.e., "row," constitute a valid codeword in the dispersal code. By means of the dispersal code, we reduce the overall storage cost of our previous construction from $n|F|$ to $(n/\ell)|F|$.

Use of a dispersal code does reduce the number of epochs T over which it is possible to ensure the integrity of F with high probability. This is because the adversary can now corrupt a given "row" / codeword merely by corrupting at least $(d-1)/2$ blocks, where d is the minimum distance of the dispersal code. (For an (n, ℓ)-Reed-Solomon dispersal code, for instance, $d = n - \ell + 1$.) In our next construction, however, we show how to reduce vulnerability to creeping-corruption attacks considerably using cryptographic integrity checks. This improvement greatly extends the integrity lifetime T of the file F.

Remark. The three simple constructions we have shown thus far have the attractive property of being *publicly verifiable*. It may be that F is encrypted and that the server code is cryptographically keyed (for reasons we explain below). Thus only the client that stored F can retrieve it. But it is still possible for *any* entity to perform an integrity check on F. Integrity checks only involve verification of block consistency across servers, and therefore don't require any secret keys. In our next construction, we sacrifice public verifiability in favor of a much longer lifetime T of integrity assurance for F.

Embedding MACs into dispersal code.

We now show how to address the problem of creeping-corruption attacks. Our solution is to authenticate matrix rows with a *message-authentication code* (MAC), computed with a secret key known by the client. A simple approach is to attach a MAC to each file block on each server. We achieve a solution with lower storage overhead than this simple approach.

Our key insight (inspired by ideas of Schwartz and Miller [31]) is to embed MACs in the parity blocks of the dispersal code. As we show, both MACs and parity blocks can be based on a universal hash function. Consequently, it is possible to create a block that is simultaneously *both* a MAC and a parity block. One of our main contributions is a construction based on this idea that we call an *integrity-protected error-correcting code* (IP-ECC) and whose details are given in Section 5.4. By inserting MACs into each row of the encoded matrix, we are able to effectively verify the responses received from servers. This mechanism protects against creeping-corruption attacks because it does not just check that rows are self-consistent as in the simpler approaches described above. Instead, with MACs, it is possible to ensure that rows *do not differ from their original values in F*.

Aggregating responses.

While the client could check individual blocks in the encoded file, a more efficient approach is to check multiple blocks of the file *simultaneously*. Another contribution of our paper is to provide a mechanism to aggregate MACs across multiple blocks. The client can specify multiple positions in the file, and verify their correctness via a single, composite response from each server.

We propose to use a linear code in HAIL called the *aggregation code* for combining servers' responses in a challenge-response protocol. The aggregate response is a linear combination of rows of the encoded file matrix, and is a codeword (or sufficiently close to a codeword) in the dispersal code. However, we need to ensure that by aggregating MAC values on individual blocks, we obtain a valid MAC. We define the notion of *composite MAC* in Section 5.3 that, intuitively, guarantees that a MAC on a vector of messages can not be obtained unless all the MACs of individual vector components are known. Note that the aggregation code in HAIL *carries zero storage overhead*: It is computed on the fly.

We describe the full HAIL system in detail in Section 6, after defining the adversarial model in Section 4. The necessary cryptographic building blocks can be found in Section 5.

4. ADVERSARIAL MODEL

We model HAIL as a set of n servers, $S_1, S_2, \ldots, S_n$, and a trusted, external entity $\mathcal{T}$. We assume authenticated, private channels between $\mathcal{T}$ and each server. In practice $\mathcal{T}$ may be a client or an external auditor.

We consider an adversary $\mathcal{A}$ that is *mobile*, i.e., can corrupt a different set of servers in each epoch, and is *Byzantine*, i.e., can behave arbitrarily. Obviously, meaningful file availability is not possible against a fully Byzantine adversary that controls *all* servers. Consequently, we assume that our adversary controls *at most* b servers in any given epoch.

We regard each server S_i as containing a distinct *code base* and *storage system*. The code base determines how the server replies to challenges; the storage system contains a (potentially corrupted) file segment.

At the beginning of each epoch, $\mathcal{A}$ may choose a fresh set of b servers and arbitrarily corrupt both their code bases and storage systems. At the end of an epoch, however, we assume that the code base of every server is restored to a correct state. From a theoretical perspective, this restoration models the limitation of the adversary to b servers. From a practical perspective, code-base restoration might reflect a malware-detection pass, software re-installation, invocation of a fresh virtual machine image, etc. Even when the code base of a server is restored, however, the adversary's corruptions to the server's storage system remain.

Repair of servers' storage systems only happens when a client reactively invokes the redistribute function—an expensive and generally rare event. Thus, while the adversary controls only b servers, it is possible for more than b servers to contain corrupted data in a given epoch. The aim of the client in HAIL is to detect and repair corruptions before they render a file F unavailable.

A time step or epoch in HAIL thus consists of three phases:

1. A *corruption* phase: The adversary $\mathcal{A}$ chooses a set of up to b servers to corrupt (where b is a security parameter).

2. A *challenge* phase: The trusted entity $\mathcal{T}$ challenges some or all of the servers.

3. A *remediation* phase: If $\mathcal{T}$ detects any corruptions in the challenge phase, it may modify / restore servers' file shares.

Let F denote the file distributed by $\mathcal{T}$. We let $F_t^{(i)}$ denote the file share held by server S_i at the beginning of epoch t, i.e., prior to the corruption phase, and let $\hat{F}_t^{(i)}$ denote the file share held by S_i after the corruption phase.

4.1 HAIL: Formal preliminaries

In our formal adversarial model, we let a system HAIL consist of the following functions:

• $\mathsf{keygen}(1^\lambda) \rightarrow \kappa$: Generates a key $\kappa = (sk, pk)$ of size security parameter λ. (For symmetric-key systems, pk may be null.)

• $\mathsf{encode}(\kappa, F, \ell, n, b) \rightarrow \{F_0^{(i)}\}_{i=1}^n$: Encodes F as a set of file segments, where $F_0^{(i)}$ is the segment designated for server i. The encoding is designed to provide ℓ-out-of-n redundancy across servers and to provide resilience against an adversary that can corrupt at most b servers in any time step.

• $\mathsf{decode}(\kappa, t, \{\hat{F}_t^{(i)}\}_{i=1}^n) \rightarrow F$: Recovers the original file F at time t from a set of file segments stored at different servers.

• $\mathsf{challenge}(\kappa) \rightarrow \{C_i\}_{i=1}^n$: Generates a challenge value C_i for each server i.

• $\mathsf{respond}(i, C_i, \hat{F}_t^{(i)}) \rightarrow R_i$: Generates server's S_i response at time t to challenge C_i.

• $\mathsf{verify}(\kappa, j, \{C_i, R_i\}_{i=1}^n) \rightarrow \{0, 1\}$. Checks the response of server j, using the responses of all servers $R_1, \ldots, R_n$ to challenge set $C_1, \ldots, C_n$. It outputs a '1' bit if verification succeeds, and '0' otherwise. We assume for simplicity that verify is sound, i.e., returns 1 for any correct response.

• $\mathsf{redistribute}(\kappa, t, \{\hat{F}_t^{(i)}\}_{i=1}^n) \rightarrow \{F_{t+1}^{(i)}\}_{i=1}^n \cup \perp$: Is an interactive protocol that replaces the fragment $\hat{F}_t^{(i)}$ stored at server i with $F_{t+1}^{(i)}$. It implements a recreation and distribution of corrupted file segments, and outputs $\perp$ if the file can not be reconstructed.

4.2 Security model: Formalization

The adversary $\mathcal{A}$ is assumed to be stateful and have access to oracles encode and verify; we assume that $\mathcal{A}$ respects the bound b on the number of permitted corruptions in a given epoch. Denote by π the system parameters $(\ell, n, b, T, \epsilon_q, n_q)$.

$\mathcal{A}$ participates in the two-phase experiment in Figure 2. In the test phase, $\mathcal{A}$ outputs a file F, which is encoded and distributed to servers. The second phase is a challenge phase that runs for T epochs. In each epoch, $\mathcal{A}$ is allowed to corrupt the code base and storage system of at most b out of n servers. Each server is challenged n_q times in each epoch, and $\mathcal{A}$ responds to the challenges sent to the corrupted servers. If more than a fraction ϵ_q of a server's responses are incorrect, the redistribute algorithm is invoked.

After the experiment runs for T time intervals, a decoding of the file is attempted and the experiment outputs 1 if the file can not be correctly recovered. We define the HAIL-advantage of $\mathcal{A}$ as: $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi) = 1]$.

Remark. In the POR security definition, by analogy with zero-knowledge proofs, the same interface used for challenge-response interactions between the client and server is also available for file *extraction*. In the POR model, the (single) server is permanently controlled by the adversary. In contrast, in HAIL only at most b out of the n servers can be corrupted in one time epoch. We could

```
Experiment $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi)$:
  $\kappa = (sk, pk) \leftarrow \mathsf{keygen}(1^\lambda)$
  $F \leftarrow \mathcal{A}(\text{"test"}, pk, \pi)$                       /* output file $F$ */
  $\{F_0^{(i)}\}_{i=1}^n \leftarrow \mathsf{encode}(\kappa, F, \ell, n, b)$          /* compute file shares */
  for $t = 0$ to $T$ do
    $A_t \leftarrow \mathcal{A}(\text{"corrupt servers"})$              /* set of corrupted servers */
    for $i = 1$ to $n$ do
      $V_i \leftarrow 0$                             /* number of correct replies for $S_i$ */
    for $a = 1$ to $n_q$ do                              /* generate $n_q$ challenges */
      $C = (C_1, \ldots, C_n) \leftarrow \mathsf{challenge}(\kappa)$
      for $j = 1$ to $n$ do                              /* challenge all servers */
        if $j \in A_t$ then                                 /* $\mathcal{A}$ responds for
          $R_j \leftarrow \mathcal{A}(\text{"respond"}, C_j, \hat{F}_t^{(j)})$               corrupted servers */
        else $R_j \leftarrow \mathsf{respond}(j, C_j, \hat{F}_t^{(j)})$
      for $j = 1$ to $n$ do                              /* verify all responses */
        if $\mathsf{verify}(\kappa, j, \{C_i, R_i\}_{i=1}^n) = 1$ then
          $V_j \leftarrow V_j + 1$                          /* $S_j$ replied correctly */
    $\mathsf{S_{corr}} \leftarrow \Phi$          /* servers with small fraction of incorrect replies */
    for $j = 1$ to $n$ do                  /* compute fraction of correct replies */
      if $\frac{V_j}{n_q} \geq 1 - \epsilon_q$ then
        $\mathsf{S_{corr}} \leftarrow \mathsf{S_{corr}} \cup \{j\}$        /* $S_j$'s incorrect replies below $\epsilon_q$ */
    if $\mathsf{S_{corr}} = \{1, 2, \ldots, n\}$ then
      $\{F_{t+1}^{(i)}\}_{i=1}^n \leftarrow \{\hat{F}_t^{(i)}\}_{i=1}^n$            /* shares remain the same */
    else $\{F_{t+1}^{(i)}\}_{i=1}^n \leftarrow \mathsf{redistribute}(\kappa, t, \{\hat{F}_t^{(i)}\}_{i=1}^n)$
  if $\mathsf{decode}(\kappa, T, \{\hat{F}_T^{(i)}\}_{i=1}^n) = F$ output 0   /* $F$ can be recovered */
  else output 1                                   /* $F$ is corrupted */
```

Figure 2: HAIL security experiment.

construct a stronger security model for HAIL in which the file could be extracted through the challenge-response protocol if decoding fails. However, the stronger model would only benefit in extracting of file fragments for those b servers corrupted by an adversary in an epoch (the other $n - b$ servers have a correct code base). We do not investigate this model further in the paper.

5. BUILDING BLOCKS

We introduce the main technical building blocks of HAIL. Proofs for all claims can be found in the full version of the paper [5].

5.1 UHFs and Reed-Solomon codes

Let I denote a field with operations $(+, \times)$. For example, in our prototype implementation, we work with $GF[2^{128}]$.

A UHF [9] is an algebraic function $h : \mathcal{K} \times I^\ell \rightarrow I$ that compresses a message $m \in I^\ell$ into a compact digest based on a key $\kappa \in \mathcal{K}$ such that the hash of two different messages is different with overwhelming probability over keys. A related notion is that of almost exclusive-or universal (AXU) hash functions. Formally:

Definition 1. *h is an ϵ-universal hash function family if for any $x \neq y \in I^\ell$: $Pr_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) = h_\kappa(y)] \leq \epsilon$.*

h is an ϵ-AXU family if for any $x \neq y \in I^\ell$, and for any $z \in I$: $Pr_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) \oplus h_\kappa(y) = z] \leq \epsilon$.

Many common UHFs are also linear, meaning that for any message pair (m_1, m_2), it is the case that $h_\kappa(m_1) + h_\kappa(m_2) = h_\kappa(m_1 + m_2)$. In fact, it is possible to construct a UHF based on a linear error-correcting code (ECC). An (n, ℓ, d) ECC encodes messages of length ℓ into codewords of size n such that the minimum distance between any two codewords is d. An (n, ℓ, d) code can correct up to $d - 1$ errors and $\lfloor \frac{d-1}{2} \rfloor$ erasures.

For example, a UHF may be based on a $(n, \ell, n - \ell + 1)$-Reed-Solomon code over I. Let $\vec{m} = (m_1, m_2, \ldots, m_\ell)$, where $m_i \in I$. $\vec{m}$ may be viewed in terms of a polynomial representation of the form $p_{\vec{m}}(x) = m_\ell x^{\ell-1} + m_{\ell-1} x^{\ell-2} + \ldots + m_1$. A Reed-Solomon code, then, may be defined in terms of a vector $\vec{a} = (a_1, \ldots, a_n)$. The codeword of a message $\vec{m}$ is the evaluation of polynomial $p_{\vec{m}}$ at points $(a_1, \ldots, a_n)$: $(p_{\vec{m}}(a_1), p_{\vec{m}}(a_2), \ldots, p_{\vec{m}}(a_n))$.

A UHF of interest, then, is simply $h_\kappa(m) = p_{\vec{m}}(\kappa)$ with key space $\mathcal{K} = I$. It is well known that this construction, denoted RS-UHF (and typically referred as the *polynomial evaluation* UHF), is indeed a good UHF [34]:

Fact 1. RS-UHF *is a $\frac{\ell-1}{2^\alpha}$-universal hash family (and, as such, a $\frac{\ell-1}{2^\alpha}$-AXU family).*

5.2 MACs obtained from UHFs

A UHF, however, is not a cryptographically secure primitive. That is, it is not generally collision-resistant against an adversary that can choose messages after selection of κ. Thus a UHF is not in general a *message-authentication code (MAC)*. A MAC is formally defined as:

Definition 2. *A Message Authentication Code* $\mathsf{MA} = (\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer})$ *is given by algorithms: $\kappa \leftarrow \mathsf{MGen}(1^\lambda)$ generates a secret key given a security parameter; $\tau \leftarrow \mathsf{MTag}_\kappa(m)$ computes a tag on message m with key κ; $\mathsf{MVer}_\kappa(m, \tau)$ outputs 1 if τ is a valid tag on m, and 0 otherwise. For adversary $\mathcal{A}$, we define:* $\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{uf\text{-}mac}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{MGen}(1^\lambda); (m, \tau) \leftarrow \mathcal{A}^{\mathsf{MTag}_\kappa(\cdot), \mathsf{MVer}_\kappa(\cdot,\cdot)} : \mathsf{MVer}_\kappa(m, \tau) = 1 \wedge m$ *not tagged before*$]$.

We denote by $\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{uf\text{-}mac}}(q_1, q_2, t)$ the maximum advantage of all adversaries making q_1 queries to MTag*, q_2 queries to* MVer *and running in time at most t.*

It is well known that a MAC may be constructed as the straightforward composition of a UHF with a *pseudorandom function* (PRF) [35, 24, 30, 34]. A PRF is a keyed family of functions $g : \mathcal{K}_{\mathsf{PRF}} \times D \rightarrow R$ that is, intuitively, indistinguishable from a random family of functions from D to R.

We define the prf-advantage of an adversary $\mathcal{A}$ for family g as $\mathsf{Adv}_g^{\mathsf{prf}}(\mathcal{A}) = |\Pr[\kappa \leftarrow \mathcal{K}_{\mathsf{PRF}} : \mathcal{A}^{g_\kappa(\cdot)} = 1] - \Pr[f \leftarrow \mathcal{F}^{D \rightarrow R} : \mathcal{A}^{f(\cdot)} = 1]|$, where $\mathcal{F}^{\mathcal{D} \rightarrow \mathcal{R}}$ is the set of all functions from D to R. We denote by $\mathsf{Adv}_g^{\mathsf{prf}}(q, t)$ the maximum prf-advantage of an adversary making q queries to its oracle and running in time t.

Given a UHF family $h : \mathcal{K}_{\mathsf{UHF}} \times I^\ell \rightarrow I$ and a PRF family $g : \mathcal{K}_{\mathsf{PRF}} \times \mathcal{L} \rightarrow I$, we construct the MAC $\mathsf{UMAC} = (\mathsf{UGen}, \mathsf{UTag}, \mathsf{UVer})$ such as: $\mathsf{UGen}(1^\lambda)$ generates key (κ, κ') uniformly at random from $\mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}}$; $\mathsf{UTag} : \mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}} \times I^\ell \rightarrow \mathcal{L} \times I$ is defined as $\mathsf{UTag}_{\kappa,\kappa'}(m) = (r, h_\kappa(m) + g_{\kappa'}(r))$; $\mathsf{UVer} : \mathcal{K}_{\mathsf{UHF}} \times \mathcal{K}_{\mathsf{PRF}} \times I^\ell \times \mathcal{L} \times I$ is defined as $\mathsf{UVer}_{\kappa,\kappa'}(m, (c_1, c_2)) = 1$ if and only if $h_\kappa(m) + g_{\kappa'}(c_1) = c_2$. The tagging algorithm of UMAC outputs, in addition to the composition of UHF and PRF, a unique counter $r \in \mathcal{L}$ incremented at each invocation. Thus, the UMAC is stateful and its properties are as follows [34].

Fact 2. *Assume that h is an ϵ^{UHF}-AXU family of hash functions and g is a PRF family. Then* UMAC *is a stateful MAC with advantage:* $\mathsf{Adv}_{\mathsf{UMAC}}^{\mathsf{uf\text{-}mac}}(q_1, q_2, t) \leq \mathsf{Adv}_g^{\mathsf{prf}}(q_1 + q_2, t) + \epsilon^{\mathsf{UHF}} q_2$.

Remark. For the composition of a UHF and PRF to be a MAC, it is important that the nonces used as input into the PRF be *unique*. In our HAIL implementation, when computing the MAC for a file block, we use as input to the PRF a hash of the file name and the block offset in the file instead of a counter.

5.3 Aggregating MACs

In our HAIL protocol, we aggregate MACs on a set of file blocks for bandwidth efficiency. We define here generic composite MAC algorithms that apply to any MAC outputing tags in a field.

Let $\mathsf{MTag} : \mathcal{K} \times J \rightarrow N$ be the tagging algorithm of a MAC $\mathsf{MA} = (\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer})$ defined on messages from field J that outputs tags in a field N. Let $\vec{M} = (m_1, \ldots, m_v) \in J^v$ be a vector of messages and let $\vec{A} = (\alpha_1, \ldots, \alpha_v) \in J^v$ be a vector of scalar values with $\alpha_i \neq 0$. We define $\tau = \sum_{i=1}^{v} \alpha_i \mathsf{MTag}_\kappa(m_i)$ as the *composite* MAC of $\vec{M}$ for coefficients $\alpha_1, \ldots, \alpha_v$. If τ is as above, we define the composite MAC verification algorithm $\mathsf{CMVer}_\kappa(\{m_i, \alpha_i\}_{i=1}^{v}, \tau)$ to output 1.

Consider an adversary that has access to MTag and CMVer oracles. Intuitively, a *composite MAC* has the property that the adversary can generate a vector of messages and a composite MAC with small probability if it does not query the MTag oracle for *all* component messages of the vector.

We give a formal definition of composite MACs below, the first in the literature to the best of our knowledge.

Definition 3. *Let* MA $=$ (MGen, MTag, MVer) *be a MAC algorithm and* CMVer *the composite MAC verification algorithm defined above. For adversary $\mathcal{A}$, we define:* $\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{c\text{-}mac}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{MGen}(1^\lambda); (\{m_i, \alpha_i\}_{i=1}^{v}, \tau) \leftarrow \mathcal{A}^{\mathsf{MTag}_\kappa(\cdot), \mathsf{CMVer}_\kappa(\cdot,\cdot)} : \mathsf{CMVer}_\kappa(\{m_i, \alpha_i\}_{i=1}^{v}, \tau) = 1 \wedge \exists i \in [1, v]$ *for which m_i was not queried to* $\mathsf{MTag}_\kappa(\cdot)$].

We denote by $\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{c\text{-}mac}}(q_1, q_2, t)$ *the maximum success probability of all adversaries making q_1 queries to* MTag*, q_2 queries to* CMVer *and running in time t.*

Lemma 1. *Given a MAC* MA *on field J,* MA *extended to J^v as above is a composite MAC with advantage:* $\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{c\text{-}mac}}(q_1, q_2, t) \leq v\mathsf{Adv}_{\mathsf{MA}}^{\mathsf{uf\text{-}mac}}(q_1 + vq_2 + v - 1, 0, (v+1)t)$.

We define a *linear composite MAC* to be such that a composite MAC can be verified from a linear combination of messages: $\vec{m} = \sum_{i=1}^{v} \alpha_i m_i$ (without access to individual messages $\{m_i\}_{i=1}^{v}$).

Definition 4. *A composite MAC algorithm is linear if there exists an algorithm* CMVer-Lin *such that* $\mathsf{CMVer}_\kappa(\{m_i, \alpha_i\}_{i=1}^{v}, \tau) = 1$ *if and only if* $\mathsf{CMVer\text{-}Lin}_\kappa(\sum_{i=1}^{v} \alpha_i m_i, \tau) = 1$.

Lemma 2. *If the nonces input to the PRF in the* UMAC *construction are known, the composite MAC defined from* UMAC *is linear.*

5.4 An integrity-protected error-correcting code (IP-ECC)

Typically, a MAC is appended to a message. Our goal in this section is to define a cryptographic primitive that acts both as a MAC and an error-correcting (or erasure) code. Moreover, we leverage the redundancy added by the error-correcting code for constructing the MAC. Such a primitive allows efficient checking of server response in our HAIL protocol.

Definition 5. *For $n \geq \ell$, we define an (n, ℓ, d)-*integrity-protected error-correcting code *(denoted* IP-ECC*) as a tuple of algorithms* IC $=$ (KGenECC, MTagECC, MVerECC) *such that:*

- $\mathsf{KGenECC}(1^\lambda)$ *selects a random key κ from key space $\mathcal{K}$;*

- MTagECC : $\mathcal{K} \times I^\ell \rightarrow I^n$ *on input key κ and message $m \in I^\ell$ outputs an* integrity-protected codeword *$c \in I^n$ that acts as an encoding of m, and contains an integrity tag for m. The minimum (Hamming) distance between two codewords is d.*

- MVerECC : $\mathcal{K} \times I^n \rightarrow (\{I^\ell \cup \perp\}, \{0,1\})$ *on input a key κ and an integrity-protected codeword $c \in I^n$ outputs a message $m \in I^\ell$ (or $\perp$ upon decoding failure), as well as a one-bit with value 1 if c contains a valid integrity tag on m, and 0 otherwise.*

For an adversary $\mathcal{A}$, we define: $\mathsf{Adv}_{\mathsf{IC}}^{\mathsf{uf\text{-}ecc}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{KGenECC}(1^\lambda); c \leftarrow \mathcal{A}^{\mathsf{MTagECC}_\kappa(\cdot), \mathsf{MVerECC}_\kappa(\cdot)} : \mathsf{MVerECC}_\kappa(c) = (m, 1) \wedge m$ *not queried to* $\mathsf{MTagECC}_\kappa(\cdot)$].

We denote by $\mathsf{Adv}_{\mathsf{IC}}^{\mathsf{uf\text{-}ecc}}(q_1, q_2, t)$ *the maximum advantage of all adversaries making q_1 queries to* MTagECC*, q_2 queries to* MVerECC *and running in time at most t.*

Similarly, integrity-protected erasure codes can be defined.

We give now a construction of an IP-ECC code $\mathsf{ECC_d}$ based on a $(n, \ell, n - \ell + 1)$ Reed-Solomon (R-S) code. Intuitively, to tag a message, we encode it under the R-S code, and then apply a PRF to the last s code symbols (for $1 \leq s \leq n$ a parameter in the system), effectively obtaining a MAC on each of those s code symbols using the UMAC construction. A codeword is considered valid if at least one of its last s symbols are valid MACs under UMAC on its decoding m. More specifically, the IP-ECC $(n, \ell, d = n - \ell + 1)$ code construction $\mathsf{ECC_d}$ is defined as:

- $\mathsf{KGenECC}(1^\lambda)$ selects keys $\vec{\kappa} = \{\{\kappa_i\}_{i=1}^{n}, \{\kappa'_i\}_{i=n-s+1}^{n}\}$ at random from space $\mathcal{K} = I^n \times (\mathcal{K}_{\mathsf{PRF}})^s$. The security parameter λ specifies the size of I, as well as the length of the keys in $\mathcal{K}_{\mathsf{PRF}}$. The keys $\{\kappa_i\}_{i=1}^{n}$ define a Reed-Solomon code as described in Section 5.1 (they define the points at which polynomials are evaluated when constructing a codeword). The keys $\{\kappa'_i\}_{i=n-s+1}^{n}$ are used as PRF keys in the UMAC construction.

- $\mathsf{MTagECC}_\kappa(m_1, \ldots, m_\ell)$ outputs $(c_1, \ldots, c_n)$, where $c_i = \mathsf{RS\text{-}UHF}_{\kappa_i}(\vec{m}), i = [1, n-s]$ and $c_i = \mathsf{UTag}_{\kappa_i, \kappa'_i}(m_1, \ldots, m_\ell) = (r_i, \mathsf{RS\text{-}UHF}_{\kappa_i}(\vec{m}) + g_{\kappa'_i}(r_i)), i = [n - s + 1, n]$.

- $\mathsf{MVerECC}_\kappa(c_1, \ldots, c_n)$ first strips off the PRF from $c_{n-s+1}, \ldots, c_n$ as: $c'_i = c_i - g_{\kappa'_i}(r_i), i = [n - s + 1, n]$, and then decodes $(c_1, \ldots, c_{n-s}, c'_{n-s+1}, \ldots, c'_n)$ using the decoding algorithm of Reed-Solomon codes to obtain message $\vec{m} = (m_1, \ldots, m_\ell)$. If the decoding algorithm of the R-S code defined by points $\{\kappa_i\}_{i=1}^{n}$ fails (when the number of corruptions in a codeword is beyond $\lfloor \frac{d-1}{2} \rfloor$), then MVerECC outputs $(\perp, 0)$. If one of the last s symbols of $(c_1, \ldots, c_n)$ is a valid MAC on $\vec{m}$ under UMAC, MVerECC outputs $(\vec{m}, 1)$; otherwise it outputs $(\vec{m}, 0)$.

Error resilience of $\mathsf{ECC_d}$. The MVerECC algorithm in $\mathsf{ECC_d}$ needs at least one correct MAC block in order to verify the integrity of the decoded message. This implies that, even if the minimum distance of the underlying code is $d = n - \ell + 1$, the construction is resilient to at most $E - 1$ erasures, and $\lfloor \frac{E-1}{2} \rfloor$ errors, for $E = \min(d, s)$.

Lemma 3. *If* RS-UHF *is constructed from a $(n, \ell, n - \ell + 1)$-Reed-Solomon code and g is a PRF family, then the* IP-ECC *code* $\mathsf{ECC_d}$ *defined above has the following advantage:*

$$\mathsf{Adv}_{\mathsf{ECC_d}}^{\mathsf{uf\text{-}ecc}}(q_1, q_2, t) \leq 2\left[\mathsf{Adv}_{\mathsf{UMAC}}^{\mathsf{uf\text{-}mac}}(q_1, q_2, t)\right].$$

Aggregating MACs for IP-ECC *codes.* The techniques we developed in Section 5.3 for aggregating MACs, i.e., for composite MAC verification, apply in a natural way to IP-ECC codes. Consider the linear combination of IP-ECC codewords $\vec{c}_1, \ldots, \vec{c}_v$ as a *composite codeword* $\vec{c} = \sum_{i=1}^{v} \alpha_i \vec{c}_i$. Implicit in $\vec{c}$ are composite MACs, i.e., linear combinations of MACs from the individual, contributing codewords. So we can apply MVerECC directly to $\vec{c}$, thereby verifying the correctness of $\vec{c}_1, \ldots, \vec{c}_v$.

Systematic IP-ECC *codes.* In a systematic code, codewords are formed by appending parity blocks to messages. The Reed-Solomon codes obtained through polynomial evaluation are, in general, not systematic. However, it is possible to offer a different view of R-S encoding that is, in fact, systematic. The *codebook* for an R-S code specified by $\vec{a} = (a_1, \ldots, a_n)$ consists of all polynomials of degree $\ell - 1$ evaluated on $\{a_i\}_{i=1}^{n}$: $C_{RS} = \{(f(a_1), \ldots, f(a_n)) | deg(f) \leq \ell - 1\}$. A systematic code is one in which a message is mapped to a codeword whose first ℓ symbols match the message (given a message $\vec{m} = (m_1, \ldots, m_\ell)$, a unique polynomial f of degree $\ell - 1$ for which $f(a_i) = m_i, i = [1, \ell]$ can be determined).

The IP-ECC construction can be adapted for systematic Reed-Solomon codes as follows: we encode a message under a systematic code, and then apply the PRF only to the parity blocks. Our results in Lemma 3 still hold for this systematic encoding for $s = n - \ell$. We employ this systematic code that can recover from $n - \ell - 1$ erasures and $\lfloor \frac{n-\ell-1}{2} \rfloor$ errors in our HAIL protocol.

5.5 Adversarial codes

Adversarial codes [6, 23] are keyed codes resistant to a large fraction of adversarial corruptions (within classical error-correcting bounds) against a computationally bounded adversary. BJO [6] define adversarial codes formally and give the first practical systematic construction based on cryptographically protected, striped Reed-Solomon codes. We omit the formal definition of adversarial codes, but intuitively, an adversary has advantage γ for a secret-key adversarial code if she is able to output a pair of codewords at small Hamming distance that decode to different messages. We refer the reader to [6] for full details of definition. A related notion is that of *computational codes*, codes that achieve higher error resilience than classical error-correcting codes by exploiting computationally bounded channels [18, 26].

In the BJO construction, the file is permuted first with a secret key and then divided into stripes. Parity blocks are computed for each stripe and appended to the unmodified file. To hide stripe boundaries, parity blocks are encrypted and permuted with another secret key. The encoding of the file consists of the original file followed by the permuted and encrypted parity blocks, and is systematic. The same construction (without rigorous formalization, though) has been proposed independently by Curtmola et al. [10]. We employ this construction for the server code in HAIL.

6. HAIL: PROTOCOL SPECIFICATION

Using the technical building blocks defined in Section 5, in this section we give full details on the HAIL protocol.

6.1 Key Generation

Let ℓ be the number of primary servers, and n the total number of servers. The client generates the following sets of keys:

- Dispersal-code keys: These are $n-\ell$ pairs of keys $\{\kappa_j, \kappa'_j\}_{j=\ell+1}^{n}$, for the UHF and PRF in the UMAC construction given in Section 5.2, respectively;

- Server-code keys: These are n keys (one per server) for the server code described in Section 5.5; and

- Challenge keys: These are keys used to generate challenges and to seed inputs to the aggregation code for responses. They can be generated from a master key that the client stores locally.

6.2 Encoding Files

The encoding of files in HAIL has been depicted in Figure 1. We aim at obtaining a distributed, systematic encoding F_d of a file F. First, we partition F into ℓ distinct segments $F^{(1)}, \ldots, F^{(\ell)}$ and distribute these segments across the primary servers $S_1, \ldots, S_\ell$ respectively. Each segment can be viewed as a set of blocks (or symbols) with values in a field I (in our implementation we use $I = GF[2^{128}]$). This distributed cleartext representation of the file remains untouched by our subsequent encoding steps.

We then encode each segment $F^{(j)}$ under the server code (implemented with the adversarial erasure code construction of BJO [6] described in Section 5.5) to protect against small corruption at each server. The effect of the server code is to extend the "columns" of F_d by adding parity blocks. Next, we apply the dispersal code ECC$_d$ (as defined in Section 5.4) to create the parity blocks that reside on the secondary servers $S_{\ell+1}, \ldots, S_n$. It extends the "rows" of F_d across the full set of n servers. To embed the dispersal code in a full-blown IP-ECC, we also add PRF values on the parity blocks for each row. Viewed another way, we "encrypt" columns $\ell + 1$ through n, thereby turning them into cryptographic MAC values.

Finally, to allow the client to confirm when it has successfully downloaded F, we compute and store on the server a MAC over F.

The steps of encode are detailed below:

1. [File partitioning] Partition the file into ℓ segments and store segment $F^{(j)}$ on S_j, for $j = [1, \ell]$. Denote by $m_F = |F|/\ell$ the number of blocks in each segment. We have obtained a (m_F, ℓ) matrix $\{F_{ij}\}_{i=[1,m_F],j=[1,\ell]}$ containing the original file blocks.

2. [Server code application] Encode each file segment $F^{(j)}$ under the systematic server code with symbols in I (viewed as an erasure code), and obtain a segment of m blocks at each server (where blocks $m_F + 1, \ldots, m$ are parity blocks for the server code).

3. [Dispersal code application] Apply the systematic dispersal code ECC$_d$ as defined in Section 5.4 to the rows of the encoded matrix from step 2. We determine thus segments $F^{(\ell+1)}, \ldots, F^{(n)}$.
If we denote by $F_d = \{F_{ij}^d\}_{i=[1,m],j=[1,n]}$ the encoded representation of F at the end of this step, then $F_{ij}^d = F_{ij} \in I$ (i.e., block i in $F^{(j)}$), for $i = [1, m_F], j = [1, \ell]$. F_{ij}^d for $i = [m_F + 1, m], j = [1, \ell]$ are the parity blocks under the server code. The columns $\ell + 1, \ldots, n$ are obtained through the application of the ECC$_d$ construction to columns $1, \ldots, \ell$ as follows: $F_{ij}^d = \text{RS-UHF}_{\kappa_j}(F_{i1} \ldots F_{i\ell}) + g_{\kappa'_j}(\tau_{ij})$, for $i = [1, m], j = [\ell + 1, n]$. τ_{ij} is a position index that depends on the file handle, block index i and server index j, e.g., hash of the file name, i and j. RS-UHF is the universal hash function construction based on Reed-Solomon codes given in Section 5.1.

4. [Whole-file MAC computation] Lastly, a cryptographic MAC of the file (and its handle) is computed and stored with the file.

The initial share at time 0 for each server S_j is $F_0^{(j)} = \{F_{ij}^d\}_{i=1}^{m}$.

6.3 Decoding Files

For decoding the encoded matrix, there are two cases to consider:

- *If the dispersal code is an error-correcting code*, then up to $\lfloor \frac{n-\ell-1}{2} \rfloor$ errors can be corrected in each row. This choice imposes the requirement that $b \leq \lfloor \frac{n-\ell-1}{2} \rfloor$. (Otherwise, the adversary could corrupt all rows in an epoch, obliterating F).

In this case, decoding of the matrix proceeds first on rows, and then on columns. In the first step, each row of the matrix is decoded and the corresponding message is checked for integrity using the MACs embedded in the parity blocks. If a row can not be correctly decoded (i.e., the number of corruptions exceeds the error correction capability of the dispersal code) or if none of the MACs in the parity blocks of a row verifies, then we mark all blocks in that row as erasures. In the second step, the server code implemented with an erasure code is used to recover the row erasures from the first step.

- *If the dispersal code is an erasure code*, the protocol tolerates up to $b \leq n - \ell - 1$ failures per epoch. In this case, we could employ an error-correcting server code. Decoding proceeds first on columns, to recover from small corruptions within each server. Then, the rows of the matrix are corrected with the dispersal erasure code.

A mechanism for determining the positions of errors in a row is needed. We can find erroneous blocks using the embedded MACs on the parity blocks, as long as at least one of the MACs in the parity blocks is valid. This approach requires brute force: We consider in turn each MAC block to be valid, try all sets of ℓ blocks in the codeword (among the $n-1$ remaining blocks), until we find a decoding for which the MAC block is valid. The brute force approach can recover from $n-\ell-1$ erasures.

Using an erasure code instead of an error-correcting code for $\mathsf{ECC_d}$ requires fewer secondary servers. The required brute-force decoding, though, is asymptotically inefficient, since $(n-\ell)\binom{n-1}{\ell}$ combinations of blocks have to be examined. In the rest of the paper, we assume that the dispersal code is an error-correcting code. Nonetheless, we can construct protocols for erasure dispersal codes.

6.4 The Challenge-Response Protocol

In the HAIL challenge-response protocol, the client verifies the correctness of a random subset of rows $D = i_1, \ldots, i_v$ in the encoded matrix. The client's challenge consists of a seed κ_c from which each server derives set D, as well as a value $u \in I$.

Each server S_j returns a linear combination of the blocks in the row positions of D, denoted by R_j. To aggregate server responses, we use an aggregation code $\mathsf{ECC_a}$ with message size v, implemented also with a Reed-Solomon code. R_j is computed as the u^{th} symbol in $\mathsf{ECC_a}$ across the selected rows. The responses of all servers $(R_1, \ldots, R_n)$ then represent a linear combination of rows $i_1, \ldots, i_v$ with coefficients $\alpha_i = u^{i-1}, i = [1, v]$.

Intuitively here, because all servers operate over the same subset of rows D, the sequence $R = (R_1, \ldots, R_n)$ is itself a codeword in the dispersal code—with aggregate PRF pads "layered" onto the responses $R_{\ell+1}, \ldots, R_n$ of the parity servers. Thanks to our IP-ECC dispersal code and our techniques of aggregating several MACs into a composite MAC (described in Section 5.3), the client can check the validity of the combined response R, by decoding to a message $\vec{m}$ and checking that at least one of the (composite) responses R_j of the secondary servers is a valid (composite) MAC on $\vec{m}$. Having done so, the client can then check the validity of each *individual* response R_j: R_j is a valid response for a primary server if it matches the j-th symbol in $\vec{m}$; for a secondary server, R_j is a valid response if it is a valid MAC on $\vec{m}$.

The challenge-response protocol is described below:

1. The client sends a challenge κ_c to all servers.

2. Upon receiving challenge κ_c, server S_j derives set $D = \{i_1, \ldots, i_v\}$, as well as a value $u \in I$. The response of server S_j is $R_j = \mathsf{RS\text{-}UHF}_u(F^d_{i_1 j}, \ldots, F^d_{i_v j})$.

3. The client calls the linear composite $\mathsf{MVerECC}$ algorithm of the dispersal code (as described in Section 5.4) on $(R_1, \ldots, R_n)$. If the algorithm outputs $(\vec{m}, 0)$ or $(\perp, 0)$, then verification of the response fails and $\mathsf{verify}(\kappa, j, \{\kappa_c, R_i\}_{i=1}^n)$ returns 0 for all j.

4. Otherwise, let $(\vec{m}, 1)$ be the output of the composite $\mathsf{MVerECC}$ algorithm. Algorithm $\mathsf{verify}(\kappa, j, \{\kappa_c, R_i\}_{i=1}^n)$ returns 1 if:

- $m_j = R_j$, for $j \in [1, \ell]$; or

- R_j is a valid composite MAC on $\vec{m}$ under UMAC with keys (κ_j, κ'_j) and coefficients $\{\alpha_i\}_{i=1}^v$, for $j \in [\ell+1, n]$.

As an optimization, the client can first check that the responses $(R_1, \ldots, R_n)$ are valid without involving the algorithm $\mathsf{MVerECC}$ of the dispersal code. To do so, the client computes the valid codeword of the first ℓ positions $(R_1, \ldots, R_\ell)$ from the vector of responses. If at least one parity block in this codeword matches the received response, the client has found the correct message without involving the expensive decoding algorithm of Reed-Solomon codes used in $\mathsf{MVerECC}$. In this case, the client could skip step 3, and proceed directly to step 4 in the above algorithm.

As in Section 6.3, a brute force approach to the decoding step in $\mathsf{MVerECC}$ could be applied if the dispersal code is an erasure code, instead of an error-correcting code.

6.5 Redistribution of Shares

HAIL runs for a number of epochs T. In each epoch the client issues n_q challenges to all servers and verifies their responses. The client monitors all servers in each epoch, and if the fraction of corrupted challenges in at least one server exceeds a threshold ϵ_q, the $\mathsf{redistribute}$ algorithm is called.

In the $\mathsf{redistribute}$ algorithm, the client downloads the file shares of all servers, and applies the decoding algorithm described above. Once the client decodes the original file, she can reconstruct the shares of the corrupted servers as in the original encoding algorithm. The new shares are redistributed to the corrupted servers at the beginning of the next time interval $t+1$ (after the corruption has been removed through a reboot or alternative mechanism). Shares for the servers that have correct shares remain unchanged for time $t+1$. We leave the design of more efficient $\mathsf{redistribute}$ algorithms for future work.

7. SECURITY ANALYSIS

We define the HAIL system to be *available* if the experiment from Figure 2 outputs 0; otherwise we say that the HAIL system is *unavailable*. HAIL becomes unavailable if the file can not be recovered either when a $\mathsf{redistribute}$ is called or at the end of the experiment. In this section, we give bounds for HAIL availability and show how to choose parameters in HAIL for given availability targets. Full proofs are deferred to the full version of the paper [5].

There are several factors that contribute to HAIL availability. First is the redundancy embedded in each server through the server code; it enables recovery from a ϵ_c fraction of corruption at each server. Second is the frequency with which the client challenges each server in an epoch; this determines the probability of detecting a corruption level greater than ϵ_c at each server. Third, the redundancy embedded in the dispersal code enables file recovery even if a certain threshold of servers are corrupted.

Challenge frequency. In HAIL, n_q challenges are issued by the client in an epoch. A $\mathsf{redistribute}$ operation is triggered if at least one of the servers replies incorrectly to more than a ϵ_q-fraction of challenges. Recall that at least $n-b$ servers have a correct code base in a time interval, but might have corruptions in their storage system. We refer to these corruptions as *residual*—they were "left behind" by $\mathcal{A}$. We are interested in detecting servers whose residual corruptions exceed the correction level ϵ_c tolerated by the server code.

Given a ϵ_c fraction of residual corrupted blocks from a server's fragment, we can compute a lower bound on the fraction of challenges that contain at least one incorrect block $\epsilon_{q,c} = 1 - \frac{\binom{(1-\epsilon_c)m}{v}}{\binom{m}{v}}$ (for m the size of file segments and v the number of blocks aggregated in a challenge). Based on $\epsilon_{q,c}$, we can determine a threshold ϵ_q (chosen at $\frac{\epsilon_{q,c}}{2}$) at which the client considers the server fragment corrupted and calls the $\mathsf{redistribute}$ algorithm. We estimate the probability p_n that we fail to detect corruption of at least a ϵ_c-fraction of blocks.

PROPOSITION 1. *Let μ be the* $\mathsf{uf\text{-}ecc}$ *advantage of an adversary for the (composite) dispersal code* $\mathsf{ECC_d}$ *(as given by Lemmas 1 and 3 in Section 5.3 and 5.4, respectively). For $\epsilon_q = \frac{\epsilon_{q,c}}{2}$, the probability with which the client does not detect a corruption of ϵ_c fraction of blocks at a server with a correct code base is* $p_n \leq e^{-\frac{n_q(\epsilon_{q,c}-2\mu)^2}{8(\epsilon_{q,c}-\mu)}}$.

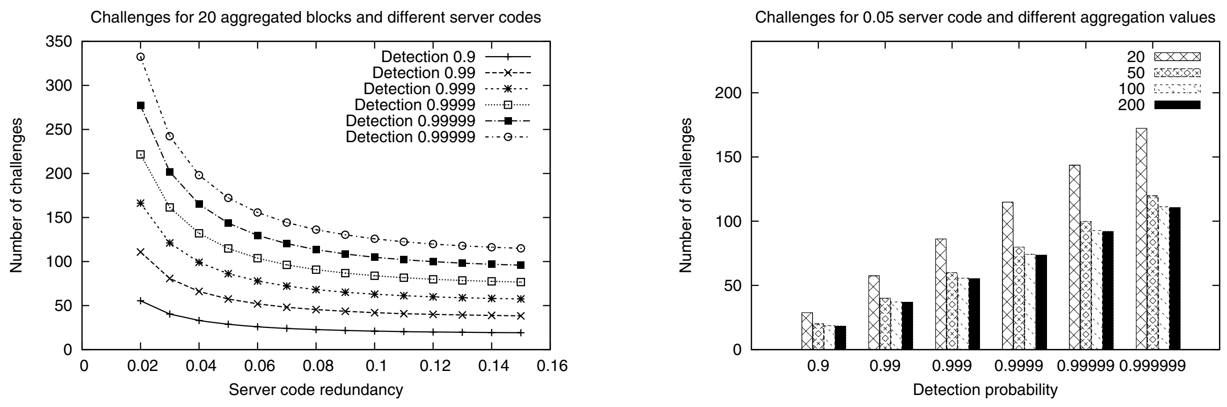


Figure 3: Number of challenges for different server codes (left) and different number of blocks aggregated in a challenge (right).

Based on the above proposition, we can choose the frequency of challenge-response interactions in an epoch based on the desired probability of detection $(1 - p_n)$, the redundancy embedded in the server code and the number of aggregated blocks in a challenge. The left graph in Figure 3 shows that the number of challenges n_q increases when the server code shrinks, and also when the detection probability increases (this graph assumes that 20 blocks are aggregated in a challenge). The right graph in Figure 3 shows that the client needs to issue less challenges in an epoch if more blocks are aggregated in a challenge (this graph is done for a server code with redundancy 5%).

Role of dispersal code. The adversary controls up to $b \leq \lfloor \frac{n-\ell-1}{2} \rfloor$ out of the n servers in epoch t and corrupted up to b servers in epoch $t-1$. Therefore, we can only guarantee that at least $n-2b$ servers successfully completed at least one challenge-response round with the client in epoch $t-1$ with a correct code base, and still have a correct code base.

For those $n-2b$ servers, there are still two cases in which a server's fragment is too heavily corrupted to be recovered with the server code: (1) The corruption level is below ϵ_c, but the server code can not correct ϵ_c—a low probability side-effect of using an "adversarial code" or (2) The corruption level is $\geq \epsilon_c$, but the HAIL challenge-response protocol didn't successfully detect the corruption. We can bound the probability of Case (1) by the adversarial code advantage γ. The probability of Case (2) is bounded above by p_n, as computed in Proposition 1.

These two bounds apply to a single server. In order to compute the availabilty of the whole HAIL system, we must treat the system as a stochastic process. Our goal, then, is to obtain an upper bound on the probability that enough fragments become unrecoverable that F is unavailable. We do so in the following theorem.

THEOREM 1. *Let U be the probability that HAIL becomes unavailable in a time epoch. Then U is upper bounded by:*

- $\left[\frac{e^\beta}{(1+\beta)^{1+\beta}}\right]^{(n-2b)(\gamma+p_n)}$, *for* $\beta = \frac{n-2b-\ell-1}{(n-2b)(\gamma+p_n)} - 1$, *if* $b < \frac{n-\ell-1}{2}$ *and* $\gamma + p_n < \frac{n-2b-\ell-1}{n-2b}$.
- $1 - [1-(\gamma+p_n)]^{\ell+1}$, *if* $b = \frac{n-\ell-1}{2}$.

The probability that HAIL becomes unavailable over an interval of t epochs is upper bounded by tU.

Figure 4 shows HAIL's availability (per epoch) for $b = 3$ faults tolerated in an epoch, different configurations for the dispersal code and different detection probabilities. In the left graph from Figure 4, the number of primary servers is fixed to 8 and the number of total servers varies from 15 to 24. In the right graph of Figure 4, the total number of servers is constant at 20 and the number of primary servers is between 6 and 13.

Consider epochs of length one week for a 2-year time interval (about 100 epochs). A 10^{-6} unavailability target for 2 years translates to 10^{-8} unavailability per epoch. This availability level can be obtained, for instance, from a (17,8) dispersal code at detection level 0.99999 or (20,9) code at detection level 0.999. Once the detection level is determined, parameters such as server code redundancy and frequency of challenge-response protocol in an epoch can be determined from Proposition 1.

Weaker adversarial model. Our experiment in Figure 2 defines a very strong adversarial model: As $\mathcal{A}$ is fully Byzantine, it can corrupt both the code base and the storage systems of servers. As servers and storage can be separate systems, it is interesting to consider a model in which the adversary only corrupts storage systems. Such a "storage-limited" adversarial model yields better security bounds: $n-b$ servers are needed to decode the file instead of $n-2b$ (under the technical condition that $n-b \geq \ell+1$). Table 1 illustrates several code parameters and the availability they offer for the weaker, "storage-limited" adversarial model.

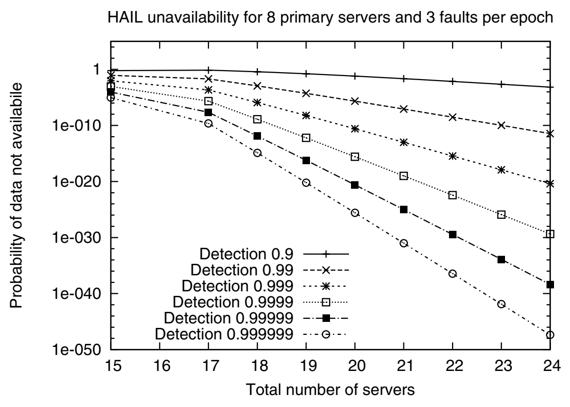
b	n	ℓ	Unavailability	b	n	ℓ	Unavailability
1	3	1	$2 \cdot 10^{-6}$	2	7	4	$5 \cdot 10^{-6}$
1	4	2	$3 \cdot 10^{-6}$	2	8	3	$6 \cdot 10^{-9}$
1	5	3	$4 \cdot 10^{-6}$	3	6	2	$3 \cdot 10^{-6}$
1	6	2	$4 \cdot 10^{-9}$	3	7	3	$3 \cdot 10^{-6}$
2	5	2	$3 \cdot 10^{-6}$	3	8	4	$5 \cdot 10^{-6}$
2	6	3	$4 \cdot 10^{-6}$	3	9	3	$6 \cdot 10^{-9}$

Table 1: Several code parameters and their availability per epoch for a weaker model.

8. IMPLEMENTATION

We have implemented HAIL file-encoding functionality in order to test the effect of dispersal code choice on encoding time. The code was written in C++ and experiments were run on an Intel Core 2 processor running at 2.16 GHz. All cryptographic operations utilize the RSA BSAFE C library.

The dispersal code was implemented using the Jerasure [29] optimized library written in C. In order to implement the integrity-protected ECC algorithm, PRF values are added to the fragments stored on secondary servers. One subtle issue when implementing the IP-ECC construction is that the symbol size of Reed-Solomon encoding should be equal to the security parameter (e.g., 128 bits). However, Jerasure implements codes with symbol sizes up to 32



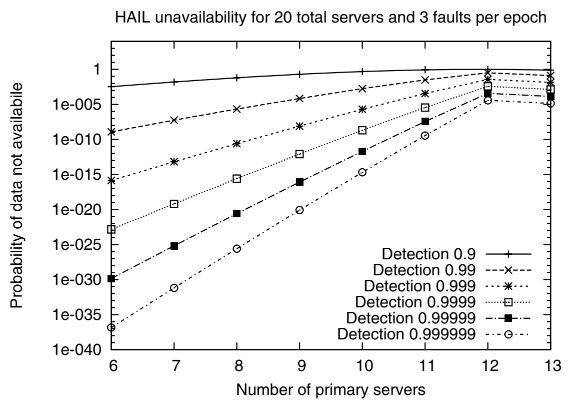
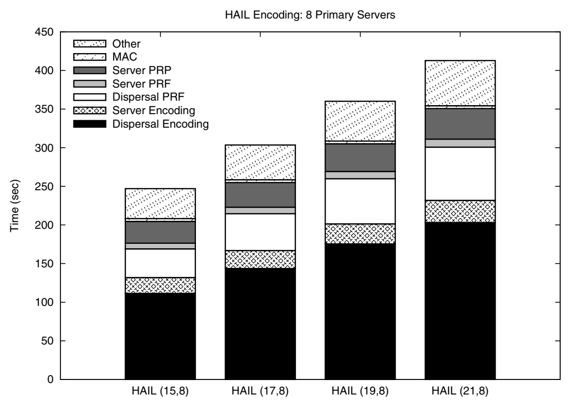


Figure 4: Probability that HAIL is unavailable for 8 primary servers (left) and 20 total servers (right) for $b = 3$ faults per epoch.



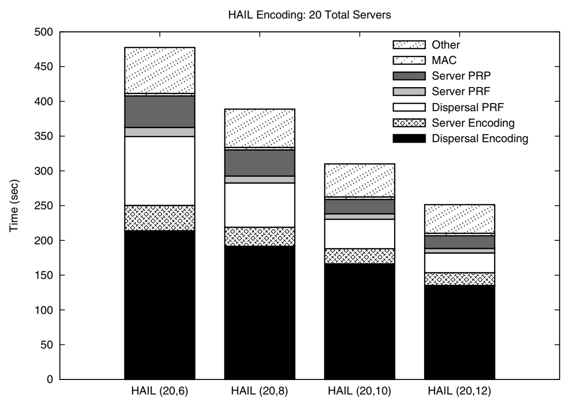


Figure 5: Encoding time for HAIL: on the left, 8 primary servers; on the right, 20 total servers.

bits. In order to obtain a UHF on 128 bits, we combine four blocks of size 32 bits, construct a polynomial of degree 4ℓ, and evaluate that polynomial four times at different random points to obtain four UHF outputs of size 32 bits.

To maximize the size of files that can be encoded efficiently using our algorithm, the file is first dispersed across the primary and secondary servers before application of the server code. Applying the server code involves a logical permutation of the file fragment using a PRP and can be done much more efficiently if each server's fragment fits into main memory. For the server code, we use a (35, 32, 3) Cauchy Reed-Solomon code over $GF[2^{32}]$ that adds 9% redundancy to data stored on each server.

Disk access is expensive and comprises 50% - 60% of our encoding time, depending on the parameters. In the graphs we present, I/O time has been removed to make the other encoding functions more visible. Figure 5 shows the encoding cost of HAIL for a 1GB file divided into several components: Jerasure dispersal code application, Jerasure server code application, the application of a PRF to the parity blocks both in the dispersal and server encoding, the time to logically rearrange the fragment on a server before application of the server code using a PRP, the computation of a MAC over the entire file, and additional data manipulations necessary to support the encoding functions. Reflecting parameter choices from Figure 4, on the left graph in Figure 5, we present the encoding cost as the number of primary servers remains constant at 8 and the total number of servers varies from 15 to 21. On the right graph in Figure 5 we keep the total number of servers constant at 20 and vary the number of primary servers between 6 and 12.

We get an encoding throughput between 2MB and 4MB per second, not including the disk I/O time. As noticed from Figure 5, time spent performing dispersal code application using Jerasure is the dominant factor in file encoding speed (at least 50% of the total encoding cost, excluding I/O). For instance, for the (20,12) dispersal code, HAIL encoding throughput is 4MB per second, compared to 7MB per second given by the dispersal code encoding. For the (21,8) dispersal code, Jerasure encoding is 5MB per second, while HAIL achieves an encoding throughput of 2.5MB per second.

As the number of secondary servers increases, the dispersal cost increases linearly, both in terms of time spent in Jerasure, as well as the time necessary to compute the required PRF values. The time spent to perform server encoding, including Jerasure application and PRP and PRF computation, increases linearly with the total amount of data to be encoded (the size of the dispersed file), which depends on both the number of primary and secondary servers.

9. CONCLUSION

We have proposed HAIL, a high-availability and integrity layer that extends the basic principles of RAID into the adversarial setting of the Cloud. HAIL is a remote-file integrity checking protocol that offers efficiency, security, and modeling improvements over straightforward multi-server application of POR protocols and over previously proposed, distributed file-availability proposals. Through a careful interleaving of different types of error-correcting layers, and inspired by proactive cryptographic models, HAIL ensures file availability against a strong, mobile adversary.

There are a number of interesting HAIL variants to explore in follow-up work. The protocols we have described above for HAIL only provide assurance for static files. We are investigating in current work design of similar protocols that accommodate file updates. We believe that the HAIL techniques we have introduced in this paper help pave the way for valuable approaches to distributed file system availability.

Acknowledgements

We thank James Hendricks, Burt Kaliski and Ron Rivest for carefully reading the paper and providing detailed comments and suggestions. We also thank Yevgeniy Dodis and Daniel Wichs for many insights on protocol design.

10. REFERENCES

[1] Amazon.com. Amazon simple storage service (Amazon S3), 2009. Referenced 2009 at aws.amazon.com/s3.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *14th ACM CCS*, pages 598–609, 2007.

[3] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession, 2008. IACR ePrint manuscript 2008/114.

[4] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO*, volume 1666 of *LNCS*, pages 216–233, 1999.

[5] K. D. Bowers, A. Juels, and A Oprea. HAIL: A high-availability and integrity layer for cloud storage, 2008. IACR ePrint manuscript 2008/489.

[6] K. D. Bowers, A. Juels, and A Oprea. Proofs of retrievability: Theory and implementation, 2008. IACR ePrint manuscript 2008/175.

[7] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *9th ACM CCS*, pages 88–97, 2002.

[8] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE SRDS*, pages 191–202, 2005.

[9] L. Carter and M. Wegman. Universal hash functions. *Journal of Computer and System Sciences*, 18(3), 1979.

[10] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *4th ACM StorageSS*, pages 63–68, 2008.

[11] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *28th IEEE ICDCS*, pages 411–420, 2008.

[12] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *6th IACR TCC*, volume 5444 of *LNCS*, pages 109–127, 2009.

[13] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *16th ACM CCS*, 2009. To appear.

[14] M. Etzel, S. Patel, and Z. Ramzan. SQUARE HASH: Fast message authentication via optimized universal hash functions. In *CRYPTO*, volume 1666 of *LNCS*, pages 234–251, 1999.

[15] D.L.G. Filho and P.S.L.M. Barreto. Demonstrating data possession and uncheatable data transfer, 2006. IACR eArchive 2006/150.

[16] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, 2000.

[17] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *34th IEEE DSN*, pages 135–144, 2004.

[18] P. Gopalan, R.J. Lipton, and Y.Z. Ding. Error correction against computationally bounded adversaries, 2004. Manuscript.

[19] S. Halevi and H. Krawczyk. MMH: Software message authentication in the Gbit/second rates. In *Fast Software Encryption*, volume 1267 of *LNCS*, pages 172–189, 1997.

[20] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *26th ACM PODC*, pages 139–146, 2007.

[21] A. Herzberg, M. Jakobsson, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *4th ACM CCS*, pages 100–110, 1997.

[22] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In *CRYPTO*, volume 1963 of *LNCS*, pages 339–352, 1995.

[23] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *14th ACM CCS*, pages 584–597, 2007.

[24] H. Krawczyk. LFSR-based hashing and authentication. In *CRYPTO*, volume 839 of *LNCS*, pages 129–139, 1994.

[25] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *USENIX Annual Technical Conference*, pages 29–41, 2003.

[26] S. Micali, C. Peikert, M. Sudan, and D. Wilson. Optimal error correction against computationally bounded noise. In *TCC*, pages 1–16.

[27] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *46th IEEE FOCS*, pages 573–584, 2005.

[28] W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In *EUROCRYPT*, volume 1233 of *LNCS*, pages 24–41, 1997.

[29] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. W. O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *7th USENIX FAST*, pages 253–265, 2009.

[30] P. Rogaway. Bucket hashing and its application to fast message authentication. In *CRYPTO*, volume 963 of *LNCS*, pages 29–42, 1995.

[31] T. J. E. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *26th IEEE ICDCS*, page 12, 2006.

[32] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, volume 5350 of *LNCS*, pages 90–107, 2008.

[33] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *11th USENIX HotOS*, pages 1–6, 2007.

[34] V. Shoup. On fast and provably secure message authentication based on universal hashing. In *CRYPTO*, volume 1109 of *LNCS*, pages 313–328, 1996.

[35] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciencies*, 22(3):265–279, 1981.