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

Kevin D. Bowers
RSA Laboratories
kbowers@rsa.com

Ari Juels
RSA Laboratories
ajuels@rsa.com

Alina Oprea
RSA Laboratories
aoprea@rsa.com

Abstract

We introduce HAIL (High-Availability and Integrity Layer), a distributed cryptographic system that permits 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.

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) [24] 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. Roughly speaking, a PDP provides weaker assurances than a POR, but potentially greater efficiency.

As a standalone tool for testing file retrievability against a single server, though, a POR is of limited value.[1] Detecting that a file is corrupted is not helpful if the file is irretrievable and thus 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 [26, 35]. 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.

[1] A standalone POR is useful for quality-of-service testing. The *speed* of the verifier's response gives an upper bound on expected delivery throughput for F. We don't treat QoS issues in this paper.

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 for distributed information dispersal algorithms such as [16, 18, 8, 21].)

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, 18, 8, 21].) The client stores just a secret key.
- *Static / dynamic file protection:* In this paper, we show how HAIL protects static stored objects, such as backup files and archives. But our tools and framework can be modified with little added overhead to accommodate file updates, i.e., to provide integrity assurance for dynamically changing objects. We briefly explain this direction in the paper conclusion.

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 [23, 22]. 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 *underengineered*: 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 proper-

ties 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 present our adversarial model in section 4 and describe technical building blocks for HAIL in section 5. We detail the HAIL protocol in section 6, and analyze its security and discuss parameter choices in Section 7. Finally, 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 file redundancy dynamically across hard-drives, HAIL manages such redundancy across cloud storage providers. Recent multi-hour failures in S3 [17] 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 in both *synchronous networks* [16], as well as *asynchronous* ones [18, 8, 21]. 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 [18, 8, 21]. In contrast, HAIL places the task of file-integrity checking in the hands of the client or some other trusted, external service and avoids communication among servers. 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* [28, 35] or *homomorphic fingerprinting* [21].) UHFs can be used to construct *message-authentication codes (MAC)* [4, 20, 14] (see [31] for a performance evaluation of various schemes). In particular, a natural combination of UHFs with pseudorandom functions (PRFs) yields MACs; these MACs can be aggregated over many data blocks and thus support compact proofs over large file samples.

PORs and PDPs Juels and Kaliski (JK) [24] propose a POR protocol and give formal security definitions. The main JK protocol supports only a limited number of challenges, whose responses are precomputed and appended to the encoded file. Shacham and Waters (SW) [36] use an implicit MAC construction that enables an unlimited number of queries, at the expense of larger storage overhead. Their MAC construction is based on the UHF + PRF paradigm, but they construct a UHF based on a random linear function, rather than a more efficient, standard error-correcting code.

In concurrent and independent work, Bowers et al. [6] and Dodis et al. [13] give general frameworks for POR protocols that generalize both the JK and SW protocols. Both papers propose the use of an error-correcting code in computing server responses to client challenges with the goal of ensuring file extraction through the challenge-response interface. The focus of [13] 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 shows how file updates might be performed in the PDP model [3]. Curtmola et al. [12] 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.

Filho and Barreto [15] describe a PDP scheme based on full-file processing. Shah et al. [37] consider a symmetric-key variant, but their scheme only works for encrypted files, and auditors must maintain long-term state. Naor and Rothblum [30], extending the memory-checking schemes of Blum et al. [5], describe a theoretical model that may be viewed as a generalization of PORs.

Distributed protocols for dynamic file-integrity checking Lillibridge et al. [26] propose a distributed scheme in which blocks of a file F are dispersed across n servers using an (n, m)-erasure code (i.e., any m out of the n fragments are sufficient to recover the file). Servers spot-check the integrity of one another's fragments using message authentication codes (MACs).

Schwartz and Miller (SM) [35] propose a scheme that ensures file integrity through distribution across multiple servers, using error-correcting codes and block-level file integrity checks. 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. Their corruption-detection system is in this view the message-authentication code (MAC) construction proposed in [39]. We adopt some ideas of simultaneous MACing and error-correcting in our HAIL constructions, but we define the construction rigorously and formally analyze its security properties.

Proactive cryptography Our adversarial model is inspired by the literature on proactive cryptography initiated by [23], which has yielded protocols resilent to mobile adversaries for secret sharing [23, 7] as well as signature schemes [22].

Proactive recovery has been proposed for the BFT system by Castro and Liskov [10]. Their system constructs a replicated state machine that tolerates a third of faulty replicas in a window of vulnerability, but any number of faults over the lifetime of the system.

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 $\lceil n/b \rceil$ epochs, the adversary will have changed F_i to $\hat{F}_i$ on *all* servers. At this point, blocks in position i will pass a consistency check by the client—even though they are corrupted.

The adversary "wins" simply by corrupting at least half

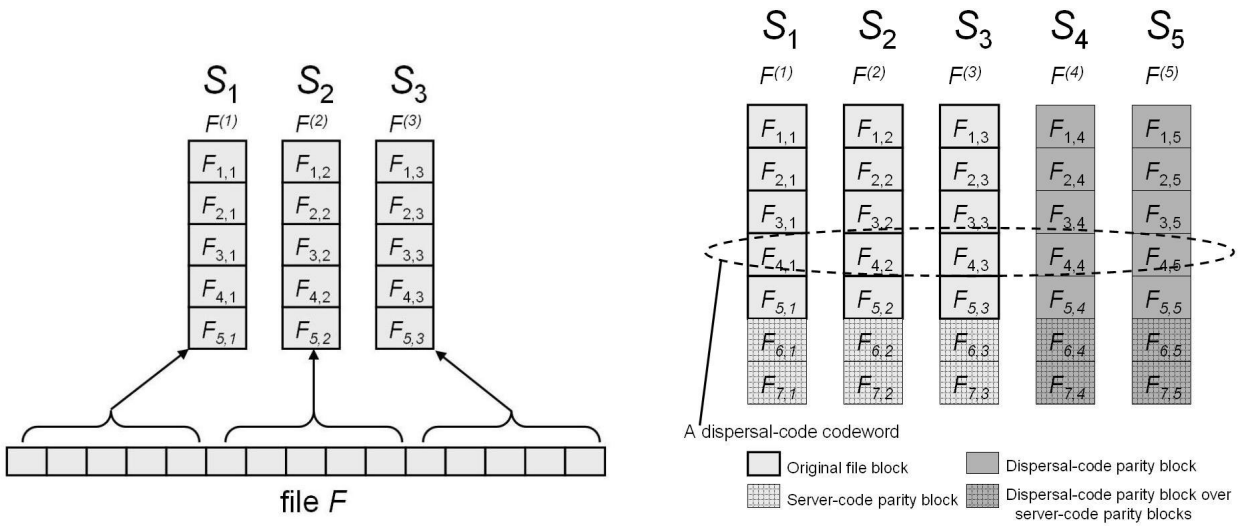

Figure 1. Encoding of file F: on the left, original file represented as a matrix; on the right, encoded file with parity blocks added for both the server and dispersal codes.

of the blocks in any position i. At that point, majority decoding will fail. The adversary can do this in $T = \lceil n/(2b) \rceil$ epochs.

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., [24, 36, 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.)

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 across the servers in any ϵ_d fraction of blocks of F with high probability, i.e., the adversary can escape detection by modifying at most ϵ_d blocks in corrupted servers.

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 $S, \ldots, S_\ell$. 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 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, however.

Our key insight (inspired by ideas of Schwartz and Miller [35]) 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). 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. (For extra bandwidth savings, the client can specify positions implicitly using a pseudorandom seed.)

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* that, intuitively, guarantees that a MAC on a vector of block 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 and the cryptographic building blocks 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 assume known upper bounds on message delivery times in the network.

We consider an adversary $\mathcal{A}$ that is *mobile*, i.e., can corrupt a different set of b 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. The adversary can of course move to a different set of corrupted servers in each 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, i.e., storage file segment, 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:

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

• $\mathsf{encode}(\kappa, F, \ell, n, b) \rightarrow \{F_0^{(i)}\}_{i=1}^n$: The function encode 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.

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

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

• $\mathsf{respond}(i, C_i, \hat{F}_t^{(i)}) \rightarrow R_i$: The function respond generates response R_i from the file fragment $\hat{F}_t^{(i)}$ stored at server i at time t to challenge C_i.

• $\mathsf{verify}(\kappa, j, \{C_i, R_i\}_{i=1}^n) \rightarrow \{0, 1\}$. The function verify checks whether the response of server j is valid, 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.

• $\mathsf{redistribute}(\kappa, t, \{\hat{F}_t^{(i)}\}_{i=1}^n) \rightarrow \{F_{t+1}^{(i)}\}_{i=1}^n \cup \perp$: The function redistribute 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. We leave the definition of redistribute as general as possible. The function may involve the client reconstructing segments from all servers, decoding F, and reinvoking encode.

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 time intervals. In each time interval, $\mathcal{A}$ is allowed to corrupt the code base and storage system of at most b out of the n servers. Each server is challenged n_q times in each interval, 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: $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi) = \Pr[\mathbf{Exp}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi) = 1]$. We denote by $\mathsf{Adv}^{\mathsf{HAIL}}(\pi, q_1, q_2, t)$ the maximum advantage of all adversaries making q_1 queries to encode, q_2 queries to verify, and running in time at most t.

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 con-

Experiment $\mathbf{Exp}^{\mathsf{HAIL}}_{\mathcal{A}}(\pi)$:
$\kappa = (sk, pk) \leftarrow \mathsf{keygen}(1^\lambda)$
$F \leftarrow \mathcal{A}(\text{“test”}, pk, \pi)$ /* 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 */
  $\mathsf{S}_{\mathsf{corr}} \leftarrow \Phi$ /* servers with small fraction of incorrect replies */
  for $j = 1$ to n do /* compute fraction of correct replies */
    if $\frac{V_j}{n_c} \geq 1 - \epsilon_q$ then
      $\mathsf{S}_{\mathsf{corr}} \leftarrow \mathsf{S}_{\mathsf{corr}} \cup \{j\}$ /* S_j's incorrect replies below ϵ_q */
  if $\mathsf{S}_{\mathsf{corr}} = \{1, 2, \ldots, n\}$ 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.

struct 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 might only benefit extraction 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 due to its complexity.

5 Building Blocks

5.1 UHFs and Reed-Solomon codes

Let I denote a field with operations $(+, \times)$. For example, in our prototype implementation, we work with $GF[2^\alpha]$ for $\alpha = 256$.

A UHF [9] is an algebraic function $h : \mathcal{K} \times I^\ell \rightarrow I$ that compresses a message or file element $m \in I^\ell$ into a compact digest or “hash” based on a key $\kappa \in \mathcal{K}$. We denote the output of h as $h_\kappa(m)$. A UHF has the property that given two inputs $x \neq y$, with overwhelming probability over keys κ, it is the case that $h_\kappa(x) \neq h_\kappa(y)$. In other words, a UHF is collision-resistant when the message pair (x, y) is selected independently of the key κ. A related notion is that of almost exclusive-or universal (AXU) hash functions that have the property that given three input messages, the probability that the XOR of the hashes of the first two inputs matches the third input is small. 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, as we assume for convenience in our work here, a UHF may be based on a $(n, \ell, d = n - \ell + 1)$-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 fixed 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, which we refer to as RS-UHF (but is typically referred as the *polynomial evaluation* UHF), is indeed a good UHF [38]:

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. Given a digest $y = h_\kappa(m)$, an adversary may be able to construct a new message m' such that $h_\kappa(m') = y$. Thus a UHF is not in general a secure *message-authentication code (MAC)*. A MAC is formally defined as follows:

Definition 2 *A Message Authentication Code (MAC) $\mathsf{MA} = (\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer})$ is given by three algorithms: $\kappa \leftarrow \mathsf{MGen}(1^\lambda)$ generates a secret key given a security parameter; $\tau \leftarrow \mathsf{MTag}_\kappa(m)$ computes a MAC on a message m with key κ; and $\mathsf{MVer}_\kappa(m, \tau)$ outputs 1 if τ is a valid MAC on m, and 0 otherwise. Consider an adversary $\mathcal{A}$ with access to the* MTag *and* MVer *oracles, whose goal is to output a valid MAC on a message not queried to* MTag*. We define:*

$\mathsf{Adv}^{\mathsf{uf\text{-}mac}}_{\mathsf{MA}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{MGen}(1^\lambda); (m, \tau) \leftarrow \mathcal{A}^{\mathsf{MTag}_\kappa(\cdot), \mathsf{MVer}_\kappa(\cdot,\cdot)} : \mathsf{MVer}_\kappa(m, \tau) = 1 \wedge m \text{ not queried to } \mathsf{MTag}_\kappa(\cdot)]$.

We denote by $\mathsf{Adv}^{\mathsf{uf\text{-}mac}}_{\mathsf{MA}}(q_1, q_2, t)$ *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) [39, 25, 34, 38]. A PRF is a keyed family of functions $g : \mathcal{K}_{\mathsf{PRF}} \times D \rightarrow R$ that maps messages from domain D to range R such that, intuitively, a random function from the PRF family is indistinguishable from a true random function from D to R.

More formally, consider an adversary algorithm $\mathcal{A}$ that participates in two experiments: one in which she has access to a function chosen randomly from family g and the second in which she has access to a random function from D to R. The goal of the adversary is to distinguish the two worlds: she outputs 1 if she believes the oracle is a function from the PRF family, and 0 otherwise.

We define the prf-advantage of $\mathcal{A}$ for family g as $\mathsf{Adv}^{\mathsf{prf}}_{g}(\mathcal{A}) = |\Pr[\kappa \leftarrow \mathcal{K}_{\mathsf{PRF}} : \mathcal{A}^{g_\kappa(\cdot)} = 1] - \Pr[f \leftarrow \mathcal{F}^{D\rightarrow R} : \mathcal{A}^{f(\cdot)} = 1]|$, where $\mathcal{F}^{\mathcal{D}\rightarrow\mathcal{R}}$ is the set of all functions from D to R. We denote by $\mathsf{Adv}^{\mathsf{prf}}_{g}(q, t)$ 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 $r \in \mathcal{L}$ that is incremented by the algorithm at each invocation. Thus, the MAC algorithm is stateful and its properties are as follows [38].

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:* $\mathsf{Adv}^{\mathsf{uf\text{-}mac}}_{\mathsf{UMAC}}(q_1, q_2, t) \leq \mathsf{Adv}^{\mathsf{prf}}_{g}(q_1 + q_2, t) + \epsilon^{\mathsf{UHF}} q_2$.

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*. One possible implementation that we have adopted in our definition above, described by Shoup [38], is to make the MAC algorithm stateful and use a counter incremented at each operation as the nonce into the PRF. In our HAIL implementation, we choose to use as the nonce input to the PRF a hash of the file name and the block offset in the file, instead of a strictly monotonic counter.

5.3 Aggregating MACs

In our HAIL protocol, we need to aggregate or compose MACs on individual file blocks into a MAC on a vector of blocks. We show in this section how we can generically compose tags computed with the same MAC algorithm. The definition of composite MAC here applies to any MAC that outputs tags in a field, not just the UMAC construction.

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 a valid composite MAC of $\{m_i\}_{i=1}^v$ for coefficients $\{\alpha_i\}_{i=1}^v$, the composite MAC verification algorithm $\mathsf{CMVer}_\kappa(\{m_i, \alpha_i\}_{i=1}^v, \tau)$ outputs 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* $\mathsf{MA} = (\mathsf{MGen}, \mathsf{MTag}, \mathsf{MVer})$ *be a MAC algorithm and* CMVer *the composite MAC verification algorithm defined above. Consider an adversary* $\mathcal{A}$ *with access to* MTag *and* CMVer *oracles whose goal is to output a set of messages* $m_1, \ldots, m_v$*, a set of coefficients* $\alpha_1, \ldots, \alpha_v$ *and a composite MAC* τ*. We define:*
$\mathsf{Adv}^{\mathsf{c\text{-}mac}}_{\mathsf{MA}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{MGen}(1^\lambda); (\{m_i, \alpha_i\}_{i=1}^v, \tau) \leftarrow \mathcal{A}^{\mathsf{MTag}_\kappa(\cdot), \mathsf{CMVer}_\kappa(\cdot,\cdot)} : \mathsf{CMVer}_\kappa(\{m_i, \alpha_i\}_{i=1}^v, \tau) = 1 \wedge \exists i \in [1, v]$ *for which* m_i *was not queried to* $\mathsf{MTag}_\kappa(\cdot)]$.

We denote by $\mathsf{Adv}^{\mathsf{c\text{-}mac}}_{\mathsf{MA}}(q_1, q_2, t)$ *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:*
$\mathsf{Adv}^{\mathsf{c\text{-}mac}}_{\mathsf{MA}}(q_1, q_2, t) \leq v\mathsf{Adv}^{\mathsf{uf\text{-}mac}}_{\mathsf{MA}}(q_1 + vq_2 + v - 1, 0, (v + 1)t)$.

Proof: Assume that there exists an adversary $\mathcal{A}$ for the composite MAC that makes q_1 queries to the tagging oracle, q_2 queries to the composite MAC verification oracle and runs in time t.

We build an adversary $\mathcal{A}'$ that targets the MAC MA. $\mathcal{A}'$ runs $\mathcal{A}$. For any query that $\mathcal{A}$ makes to the tagging oracle, $\mathcal{A}'$ returns the output of its own tagging oracle. For any query $(\{m_i, \alpha_i\}_{i=1}^v, \tau)$ that $\mathcal{A}$ makes to the composite MAC verification oracle, $\mathcal{A}'$ queries its tagging oracle v

times, once for each $m_i, i = [1, v]$, and outputs 1 if and only if the linear relation $\tau = \sum_{i=1}^{v} \alpha_i \mathsf{MTag}_\kappa(m_i)$ is verified.

Assume that $\mathcal{A}$ outputs a set of messages $(m_1^*, \ldots, m_v^*)$, a set of coefficients $(\alpha_1^*, \ldots, \alpha_v^*)$, and a valid composite MAC τ^*, such that there exists $i \in [1, v]$ for which m_i was not queried to $\mathsf{MTag}_\kappa(\cdot)$. Then, $\mathcal{A}'$ guesses i (with probability $\frac{1}{v}$), queries the tagging oracle for $m_1^*, m_{i-1}^*, m_{i+1}^*, \ldots, m_v^*$ and then outputs m_i^* and its MAC computed as $\mathsf{MTag}_\kappa(m_i^*) = (\alpha_i^*)^{-1}(\tau^* - \sum_{j \neq i} \alpha_j^* \mathsf{MTag}_\kappa(m_j^*))$.

$\mathcal{A}'$ makes $q_1 + vq_2 + (v-1)$ queries to the MAC oracle and no queries to the verification oracle. Its running time is less than $v+1$ times the running time of $\mathcal{A}$. The lemma follows. □

Sometimes we do not have access to all messages $\{m_i\}_{i=1}^v$ to check a composite MAC. We define a *linear composite MAC algorithm* to be such that a composite MAC can be verified from a linear combination of messages: $\vec{m} = \sum_{i=1}^{v} \alpha_i m_i$.

Definition 4 *A composite MAC algorithm is linear if there exists an algorithm* $\mathsf{CMVer\text{-}Lin}$ *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 nonce values r_i input to the PRF in the* UMAC *construction are known, the composite MAC defined from* UMAC *is linear.*

Proof: Assume that nonces r_i input to the PRF in the UMAC construction are known. Then we can simply view the UTag algorithm in the UMAC construction as outputting: $\mathsf{UTag}_{\kappa,\kappa'}(\vec{m}) = \mathsf{RS\text{-}UHF}_\kappa(\vec{m}) + g_{\kappa'}(r_i)$.

Let $\vec{m_1}, \ldots, \vec{m_v}$ be messages in I^ℓ, $\alpha_1, \ldots, \alpha_v$ scalar values in I, and $\vec{m} = \sum_{i=1}^{v} \alpha_i \vec{m_i} \in I^\ell$. We denote m_{ij} the j-th block in $\vec{m_i}$.

For a tag τ, we define algorithm $\mathsf{CMVer\text{-}Lin}_{\kappa,\kappa'}(\vec{m}, \tau) = 1$ if and only if $\mathsf{RS\text{-}UHF}_\kappa(\vec{m}) + \sum_{i=1}^{v} \alpha_i g_{\kappa'}(r_i) = \tau$.

We can infer that for the UMAC construction:

$$\mathsf{CMVer}_{\kappa,\kappa'}(\{m_i, \alpha_i\}_{i=1}^v, \tau) = 1 \Leftrightarrow$$
$$\sum_{i=1}^{v} \alpha_i \mathsf{UTag}_{\kappa,\kappa'}(\vec{m_i}) = \tau \Leftrightarrow$$
$$\sum_{i=1}^{v} \alpha_i [\mathsf{RS\text{-}UHF}_\kappa(\vec{m_i}) + g_{\kappa'}(r_i)] = \tau \Leftrightarrow$$
$$\sum_{i=1}^{v} \alpha_i [\sum_{j=1}^{\ell} m_{ij} \kappa^{j-1}] + \sum_{i=1}^{v} \alpha_i g_{\kappa'}(r_i) = \tau \Leftrightarrow$$
$$\sum_{j=1}^{\ell} (\sum_{i=1}^{v} \alpha_i m_{ij}) \kappa^{j-1} + \sum_{i=1}^{v} \alpha_i g_{\kappa'}(r_i) = \tau \Leftrightarrow$$
$$\mathsf{RS\text{-}UHF}_\kappa(\sum_{i=1}^{v} \alpha_i \vec{m_i}) + \sum_{i=1}^{v} \alpha_i g_{\kappa'}(r_i) = \tau \Leftrightarrow$$
$$\mathsf{CMVer\text{-}Lin}_{\kappa,\kappa'}(\vec{m}, \tau) = 1$$

□

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 code. Moreover, we leverage the redundancy added by the error-correcting code for constructing the MAC. Such a primitive proves to be important in our distributed HAIL protocol. It allows efficient checking of server response in our POR 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* $\mathsf{IC} = (\mathsf{KGenECC}, \mathsf{MTagECC}, \mathsf{MVerECC})$ *such that:*

- $\mathsf{KGenECC}(1^\lambda)$ *selects a random key κ from key space $\mathcal{K}$;*
- $\mathsf{MTagECC} : \mathcal{K} \times I^\ell \rightarrow I^n$ *takes as input a secret key κ and a message m in I^ℓ, and outputs an* integrity-protected codeword *c in space I^n that acts as an encoding of m, and contains an integrity tag for m.*
- $\mathsf{MVerECC} : \mathcal{K} \times I^n \rightarrow (\{I^\ell \cup \perp\}, \{0,1\})$ *takes as input a secret key κ and an integrity-protected codeword c in I^n and 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.*

Consider an adversary $\mathcal{A}$ with access to the $\mathsf{MTagECC}_\kappa(\cdot)$ *and* $\mathsf{MVerECC}_\kappa(\cdot)$ *oracles, whose goal is to output a codeword c such that* $\mathsf{MVerECC}_\kappa(c) = (m, 1)$ *and m was not queried to* $\mathsf{MTagECC}_\kappa(\cdot)$*. 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.*

We give now a construction of an IP-ECC code based on a $(n, \ell, n - \ell + 1)$ Reed-Solomon code, called $\mathsf{ECC_d}$. 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, ℓ, d) code construction is defined as (using notation from Sections 5.1 and 5.2):

- $\mathsf{KGenECC}(1^\lambda)$ 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 $c_i = \mathsf{UTag}_{\kappa_i,\kappa'_i}(m_1, \ldots, m_\ell) = (r_i, \mathsf{RS\text{-}UHF}_{\kappa_i}(\vec{m}) + g_{\kappa'_i}(r_i)), i = [n-s+1, n]$. We define d to be the minimum (Hamming) distance between two codewords output by MTagECC.

- $\mathsf{MVerECC}_\kappa(c_1, \ldots, c_n)$ 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, then the MVerECC algorithm outputs $(\perp, 0)$ (we assume that the decoding algorithm of R-S codes fails if the number of corruptions in a codeword is beyond $\lfloor \frac{d-1}{2} \rfloor$). If one of the last s symbols of $(c_1, \ldots, c_n)$ is a valid MAC on $\vec{m}$ under UMAC, the MVerECC algorithm outputs $(\vec{m}, 1)$; otherwise it outputs $(\vec{m}, 0)$.

Error resilience of $\mathsf{ECC_d}$. The MVerECC algorithm in the above construction needs at least one correct MAC block in order to decode and verify the message correctly. 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:*

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

Proof:

Let $\mathcal{A}$ be a successful adversary algorithm for code $\mathsf{ECC_d}$ that makes q_1 queries to the tagging oracle MTagECC, q_2 queries to the verification oracle MVerECC and runs in time t. It outputs a codeword $(c_1, \ldots, c_n)$ that decodes to message $\vec{m} = (m_1, \ldots, m_\ell)$ such that at least one of the last s symbols in the codeword is a valid MAC on $\vec{m}$ computed with algorithm UMAC defined in Section 5.2.

We build an adversary $\mathcal{A}'$ for the UMAC construction. $\mathcal{A}'$ is given access to a tagging oracle $\mathsf{UTag}_{\kappa,\kappa'}(\cdot)$ and a verification oracle $\mathsf{UVer}_{\kappa,\kappa'}(\cdot,\cdot)$ and needs to output a new message and tag pair.

$\mathcal{A}'$ chooses a position $j \in [n-s+1, n]$ at random, and generates keys $\{\kappa_i\}_{i=1}^n$ and $\{\kappa'_i\}_{i=n-s+1}^n$ for $i \neq j$. $\mathcal{A}'$ runs $\mathcal{A}$.

When $\mathcal{A}$ makes a query to tagging $\vec{m} = (m_1, \ldots, m_\ell)$, $\mathcal{A}'$ computes $c_i \leftarrow \mathsf{RS\text{-}UHF}_{\kappa_i}(\vec{m})$ for $i = [1, n-s]$, and $c_i = \mathsf{UTag}_{\kappa_i,\kappa'_i}(\vec{m})$ for $i = [n-s+1, n], i \neq j$. $\mathcal{A}'$ calls the UTag oracle to compute $c_j = \mathsf{UTag}_{\kappa,\kappa'}(\vec{m})$. $\mathcal{A}'$ then responds to $\mathcal{A}$ with $\vec{c} \leftarrow (c_1, \ldots, c_n)$.

When $\mathcal{A}$ makes a query $\vec{c} = (c_1, \ldots, c_n)$ to the verification oracle, $\mathcal{A}'$ tries to decode $(c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_n)$ into message $\vec{m}$. If decoding fails (there are more than $\lfloor \frac{E-1}{2} \rfloor$ errors in the codeword), then $\mathcal{A}'$ responds to $\mathcal{A}$ with $(\perp, 0)$. Otherwise, let $\vec{m}$ be the decoded message. $\mathcal{A}'$ makes a query to the verification oracle $a \leftarrow \mathsf{UVer}_{\kappa,\kappa'}(\vec{m}, c_j)$ and returns $(\vec{m}, a)$ to $\mathcal{A}$.

Assume that $\mathcal{A}$ outputs an encoding $\vec{c} = (c_1, \ldots, c_n)$ under $\mathsf{ECC_d}$ that can be decoded to $\vec{m}$, such that $\vec{m}$ was not an input to the tagging oracle and at least one of the last s symbols in $\vec{c}$ is a valid MAC for $\vec{m}$. Then $\mathcal{A}'$ outputs $(\vec{m}, c_j)$. Since the codeword $\vec{c}$ can be decoded, at least a majority of its parity blocks are correct. Then, with probability at least $1/2$, c_j is a correct MAC on $\vec{m}$. It follows that $\mathcal{A}'$ succeeds in outputting a correct message and MAC pair $(\vec{m}, c_j)$ with probability at least half the success probability of $\mathcal{A}$. ∎

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, the codeword for a message contains the message in clear followed by parity blocks. The Reed-Solomon codes obtained through polynomial evaluation are, in general, not systematic. However, it is possible to offer a different view of Reed-Solomon encoding that is, in fact, systematic. The *codebook* for an R-S code specified by vector $\vec{a} = (a_1, \ldots, a_n)$ consists of all polynomials of de-

gree $\ell - 1$ evaluated on each of the points of $\vec{a}$: $C_{RS} = \{(f(a_1), \ldots, f(a_n)) | deg(f) \leq \ell - 1\}$. 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 polynomial f of degree $\ell - 1$ for which $f(a_i) = m_i, i = [1, \ell]$ can be determined by solving a linear system).

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 protocols.

5.5 Adversarial codes

Adversarial ECCs [6, 24] are codes resistant to a large fraction of adversarial corruptions. While a standard ECC is designed to provide information-theoretic properties, an adversarial ECC uses cryptography to achieve otherwise inefficient (or impossible) levels of resilience against a computationally bounded adversary. Lipton [27] first proposed a model of computationally bounded channels for error-correcting codes. Subsequent papers (e.g., [19, 29]) have proposed specific constructions and analyzed their error resilience asymptotically with respect to the code rate. Ostrovsky et al. [32] construct locally decodable codes against computationally bounded noisy channels. In our work, we use cryptography to construct more *practical* error-correcting codes within classical error-correcting bounds.

An (n, ℓ, d)-error-correcting code corrects up to $\lfloor \frac{d-1}{2} \rfloor$ errors, and thus it supports a fraction of $\frac{d-1}{2n}$ adversarial corruption. But it is challenging to construct efficiently computable codes with large message sizes and strong error tolerance against an adversary. A standard technique for building ECCs with large message sizes is *striping*, an approach that encodes consecutive message chunks and then interleaves them to achieve heightened protection against, e.g., burst errors. But striping doesn't offer heightened protection against adversarial corruption: An adversary that knows the stripe structure can work around it. And while several classes of very efficient XOR erasure codes (e.g., Tornado, LT, Fountain and Raptor codes) tolerate a large fraction of randomly distributed errors, their behavior to adversarial corruptions is not understood.

BJO [6] define adversarial codes formally and give the first systematic construction based on cryptographically protected, striped Reed-Solomon codes. In their 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. [11]. We employ this construction for the server code in the HAIL protocol.

BJO define an adversarial ECC as follows. An (n, ℓ, d)-*adversarial error-correcting code* AECC consists of a public key space $\mathcal{K}_{PK}$, private key space $\mathcal{K}_{SK}$, an alphabet Σ, and a triple of functions: (1) a probabilistic function $\mathsf{KGenECC}(1^\lambda) \rightarrow (pk, sk) \in \mathcal{K}_{PK} \times \mathcal{K}_{SK}$; (2) a function $\mathsf{Encode} : \mathcal{K}_{PK} \times \mathcal{K}_{SK} \times \Sigma^\ell \rightarrow \Sigma^n$; and (3) a function $\mathsf{Decode} : \mathcal{K}_{PK} \times \mathcal{K}_{SK} \times \Sigma^n \rightarrow \{\Sigma^\ell \cup \perp\}$.

While Encode and Decode may be deterministic or probabilistic, we assume for our purposes that they are deterministic. A *secret-key* ECC is one in which $\mathcal{K}_{PK} = \phi$. We shall consider only secret-key ECCs here.

Consider an adversary $\mathcal{A}$ with access to the Encode and Decode oracles, whose goal is to output a pair of codewords at small Hamming distance that decode to different messages. We define the advantage of $\mathcal{A}$ as:

Definition 6 *Let* $\mathbf{Adv}_{\mathcal{A},\mathsf{AECC}}(\rho, \delta) = \Pr[\kappa \leftarrow \mathsf{KGenECC}(1^\lambda); (c, c') \leftarrow \mathcal{A}^{\mathsf{Decode}_\kappa(\cdot), \mathsf{Encode}_\kappa(\cdot)} : \mathsf{Decode}_\kappa(c) \neq \mathsf{Decode}_\kappa(c') \wedge \mathsf{Decode}_\kappa(c') \neq \perp] - \delta$, *where* $\mathcal{A}$ *outputs* $(c, c') \in (\Sigma^n, \Sigma^n)$ *such that: (1)* c *is the output of an oracle call* $\mathsf{Encode}_\kappa(\cdot)$ *(*c *is a valid codeword) and (2)* $|c - c'| \leq \rho n$.

The definition can be easily extended to erasure codes. It is easy to show that an (n, ℓ, d)-ECC that is an (ρ, δ)-adversarial ECC with $\rho \leq \frac{d-1}{2n}$, is also a $(2\rho, \delta)$-adversarial erasure code.

6 HAIL: Protocol Specification

Using the technical building blocks defined in the previous section, 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 $\{\kappa_j, \kappa'_j\}_{j=[\ell+1,n]}$, where κ_j and κ'_j are secret keys for the UHF and for the PRF in the UMAC construction, respectively;
- Server-code keys: These are generated via the algorithm KGenECC, described above; and

- Challenge / aggregation keys: These are keys used to generate challenges—and applied, of course, to seed inputs to the aggregation code for responses. These keys can be generated from a master secret key that the client stores locally.

6.2 Encoding Files

The encoding of files in our protocols has been depicted graphically 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. 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 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. (The dispersal code and server code commute, so we can swap these two encoding steps, if desired.) To embed the dispersal code in a full-blown $\mathsf{IP\text{-}ECC}$, we also add PRF values on the parity blocks for each row, i.e., on the blocks contained in secondary servers $S_{\ell+1}, \ldots, S_n$. Viewed another way, we "encrypt" columns $\ell+1$ through n, thereby turning them into cryptographic MAC values.

Finally, we compute a MAC over the full file F, and store this MAC value on the server. This full-file MAC allows the client to confirm when it has successfully downloaded the file F.

The steps of encode are detailed below:

1. [File partitioning] Partition the file into ℓ segments and store segment $F^{(j)}$ on server j, for $j = [1, \ell]$. Denote by $m_F = |F|/\ell$ the number of symbols in each segment. We have thus 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 server systematic server code as defined in Section 5.5 (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 $\mathsf{ECC_d}$ as defined in Section 5.4 to the rows of the encoded matrix obtained in step 2. We determine thus the segments $F^{(\ell+1)}, \ldots, F^{(n)}$.

If we denote by $F_d = \{F^d_{ij}\}_{i=[1,m],j=[1,n]}$ the encoded representation of F at the end of this step, then $F^d_{ij} = F_{ij}$ (i.e., block i in $F^{(j)}$), for $i = [1, m_F], j = [1, \ell]$. F^d_{ij} 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 $\mathsf{ECC_d}$ construction to columns $1, \ldots, \ell$ as follows: $F^d_{ij} = \mathsf{RS\text{-}UHF}_{\kappa_j}(F_{i1} \ldots F_{i\ell}) + g_{\kappa'_j}(\tau_{ij})$, for $i = [1, m], j = [\ell+1, n]$. τ_{ij} is a position index that depends on the file handle, as well as the block index i and the server position j, e.g., hash of the file name concatenated with i and j.

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 $F^{(j)}_0 = \{F^d_{ij}\}_{i=[1,m]}$.

6.3 Decoding Files

For decoding the encoded matrix, we proceed into two steps. First, we decode each row of the matrix and check the MACs embedded into the parity blocks of the dispersal code. If a row can not be correctly decoded 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, we apply the server code to each column and try to recover from the erasures in the first step. If the number of erasures exceeds the correction capability of the server code, then decoding fails.

Since the exact positions of erasures are known from the first level of decoding, the adversarial server code can be an erasure code. The dispersal code can be either an erasure code or an error-correcting code. However, each choice imposes a different requirement on the maximum number b of corrupted servers in an epoch. We present decoding algorithms for both cases.

The details of the algorithm to recover the original file from the encoded matrix F_d are as follows:

1. **Decode rows from the dispersal code.** As discussed above, we consider two cases:

   - *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 the number of servers corrupted in each epoch be bounded by $b \leq \lfloor \frac{n-\ell-1}{2} \rfloor$. (Otherwise, the adversary could corrupt all rows in an epoch, obliterating F).
   - *If the dispersal code is an erasure code*, then 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 (and thus we impose $b \leq n-\ell-1$).

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. We recommend use of an erasure code for $\mathsf{ECC_d}$ for small values of n (e.g., $n < 10$), and a full-blown error-correcting code for larger values of n. In the remainder of the paper, we assume that the dispersal code is an error-correcting code.

2. **Decode columns from the server code.** The blocks in the rows that can not be reconstructed from the dispersal code are marked as erasures in the matrix. The server code is then applied to each column to recover those erasures. If one of the columns stored at the primary servers can not be recovered, then decoding fails.

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 $j \in [\ell+1, n]$.

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 and Parameter Choices

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.

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 certain

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 uncorrectable by the server code of an individual server. Third, the redundancy embedded in the dispersal code enables file recovery even if a certain threshold of servers is corrupted.

Adversarial server erasure code. Let $(n_s, \ell_s, d_s = n_s - \ell_s + 1)$ be the parameters of the (Reed-Solomon) code used to encode stripes in the adversarial server code construction given originally in BJO [6]. As proved by BJO, the server code is an adversarial erasure code with the following advantage:

Fact 3 *If the block corruption level at a server is $\epsilon_c \leq \frac{d_s-1}{2n_s}$ and the size of a file segment stored at each server is m, then the server erasure code has advantage $\mathbf{Adv}_{\mathsf{AECC}}(\epsilon_c, \gamma)$, for $\gamma = \lceil \frac{m}{n_s} \rceil e^{d_s - 1 - 2\epsilon_c n_s} \left(\frac{d_s-1}{2\epsilon_c n_s}\right)^{1-d_s}$.*

Challenge frequency. In HAIL, n_q challenges are issued by the client in an epoch. A 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 tolerated by the server code (denoted ϵ_c).

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, denoted $\epsilon_{q,c}$. Based on $\epsilon_{q,c}$, we can then determine a threshold ϵ_q at which the client considers the server fragment corrupted and calls the redistribute algorithm. After we choose the detection threshold ϵ_q, we estimate the probability p_n that we fail to detect corruption of at least a ϵ_c-fraction of blocks.

Fact 4 *If a server's fragment contains a fraction ϵ_c of corrupted blocks and the server code base is correct, then the fraction of challenges containing a corrupted block is $\epsilon_{q,c} = 1 - \frac{\binom{(1-\epsilon_c)m}{v}}{\binom{m}{v}}$ (with m the size of server's fragment, and v the number of blocks aggregated in a challenge).*

The proof is simple. If a fraction ϵ_c of blocks are corrupted on a server, then there are $\epsilon_c m$ red (corrupted) blocks and $(1-\epsilon_c)m$ black (uncorrupted) blocks. Then, out of the space of $\binom{m}{v}$ challenges, there are $\binom{m}{v} - \binom{(1-\epsilon_c)m}{v}$ challenges containing a red block.

Proposition 1 *Let μ be the* uf-ecc *advantage of an adversary for the (composite) dispersal code $\mathsf{ECC_d}$ corrupting up to b servers in an epoch (as given by Lemmas 1 and 3). If we set $\epsilon_q = \frac{\epsilon_{q,c}}{2}$ and issue n_q challenges in an epoch, 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 $p_n \leq e^{-\frac{n_q(\epsilon_{q,c}-2\mu)^2}{8(\epsilon_{q,c}-\mu)}}$.*

Proof: Let us define a random variable $X_i = 1, i = [1, n_q]$ for each server's reply with value 1 if the reply is incorrect and 0 otherwise. Let $X = \sum_{i=1}^{n_q} X_i$ be the number of incorrect replies in an epoch. Assuming that more than a ϵ_c-fraction of blocks are corrupted on a server, then from Fact 4, $\Pr[X_i = 1] \geq \epsilon_{q,c} - \mu$ and $E(X) \geq (\epsilon_{q,c} - \mu)n_q$. If we define variables Z_i with $\Pr[Z_i = 1] = (\epsilon_{q,c} - \mu)$, and $Z = \sum_{i=1}^{n_q} Z_i$, then $Z_i \leq X_i$. We could bound the probability of not detecting corruption at threshold ϵ_c in a server with a correct code base using Chernoff bounds:
$p_n = \Pr[X < \epsilon_q n_q] < \Pr[Z < \epsilon_q n_q] = \Pr[Z < (1 - \alpha)(\epsilon_{q,c} - \mu)n_q] < e^{-\frac{(\epsilon_{q,c}-\mu)n_q\alpha^2}{2}}$, for $\alpha = \frac{\epsilon_{q,c}-2\mu}{2(\epsilon_{q,c}-\mu)}$. □

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 0.05).

Role of dispersal code. The file is distributed in HAIL to ℓ primary and $n - \ell$ secondary servers. We assume that the dispersal code is an $(n, \ell, d = n - \ell + 1)$ IP-ECC code as described in Section 5.4. The dispersal code can correct up to $\lfloor \frac{n-\ell-1}{2} \rfloor$ adversarial server corruptions in an epoch. We now look to compute an upper bound on the probability that HAIL becomes unavailable in a given time interval t.

The adversary controls up to b 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 even using 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

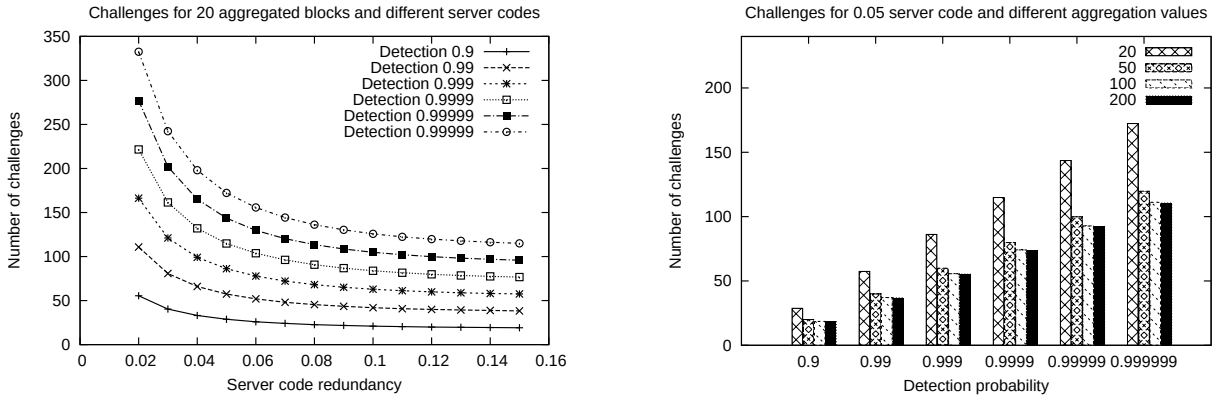

Figure 3. Number of challenges as a function of server code redundancy (left) and number of aggregated blocks in a challenge (right) for different detection probabilities.

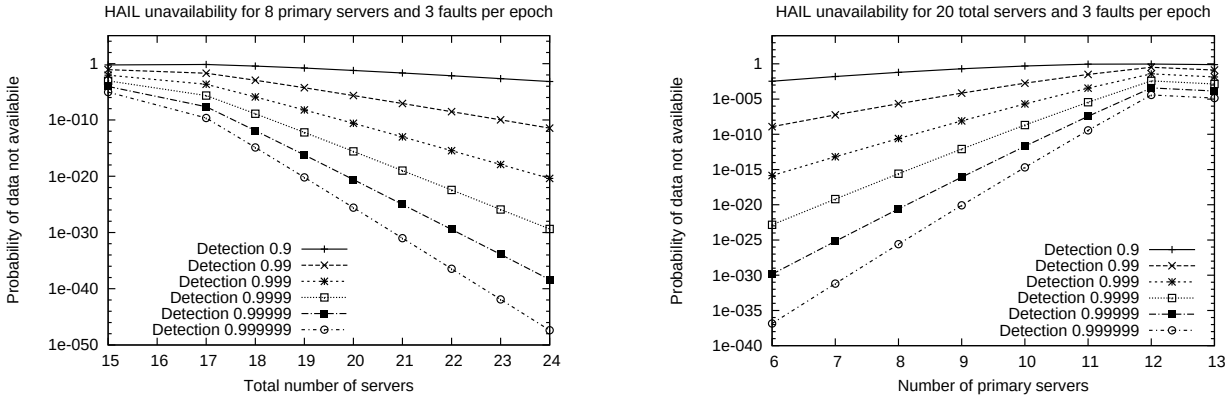

Figure 4. Probability that HAIL is unavailable for 8 primary servers (left) and 20 total servers (right) for $b = 3$ faults per epoch and different detection probabilities.

didn't successfully detect the corruption. We can bound the probability of Case (1) by γ of Fact 3. 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 proposition.

Proposition 2 *The probability U that HAIL becomes unavailable in a time epoch 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* $b = \frac{n-\ell-1}{2}$.

Proof: Assume that the HAIL system becomes unavailable at time t. Let t_0 be the last time interval before t at which a redistribute was triggered. Then, at the beginning of time interval $t_0 + 1$ all n servers have correct file shares.

The adversary corrupts at most b servers in epoch t, and at most b servers in epoch $t-1$ (that might still have residual block corruptions). The other servers (denoted $\mathsf{S_c}$) might have been corrupted at a time interval between $t_0 + 1$ and $t - 2$, but they have passed a challenge-response protocol having a correct code base in at least one time interval after they have been corrupted (since redistribute has not been triggered between time $t_0 + 1$ and $t - 1$). The size of $\mathsf{S_c}$ is at least $n - 2b$.

Let us bound HAIL unavailability using this fact. Assume, wlog, that servers $1, 2, \ldots, n - 2b$ are in $\mathsf{S_c}$. Let Y_i be a random variable for server i with value 1 if server S_i's fragment can not be recovered, and 0 otherwise. $Y_i = 1$ if either the corruption level is below ϵ_c and the adversary

has an advantage for the adversarial server code, or if the corruption level is at least ϵ_c, and it is not detected.

Then $\Pr[Y_i = 1] \leq \gamma + p_n$, and let $Y = \sum_{i=1}^{n-2b} Y_i$ with $E(Y) \leq (n-2b)(\gamma + p_n)$. Let us consider two cases:

- If $b < \frac{n-\ell-1}{2}$, then HAIL is unavailable if at least $n - 2b - \ell$ of the servers have $Y_i = 1$. We could bound HAIL unavailability using Chernoff bounds:

  $\Pr[\text{HAIL unavailable}] = \Pr[Y > n - 2b - \ell - 1] = \Pr[Y > (1+\beta)(n-2b)(\gamma + p_n)] < \left[\frac{e^\beta}{(1+\beta)^{1+\beta}}\right]^{(n-2b)(\gamma+p_n)}$.

- If $b = \frac{n-\ell-1}{2}$, then $|\mathsf{S_c}| = n - 2b = \ell + 1$ and HAIL is unavailable if the fragment from at least one of the servers in $\mathsf{S_c}$ can not be recovered. The probability of this event is $1 - [1 - (\gamma + p_n)]^{\ell+1}$.

□

Corollary 1 *The probability that HAIL becomes unavailable over an interval of t epochs is upper bounded by tU, with U given by Proposition 2.*

Figure 4 shows HAIL's availability (per epoch) for 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.

Assume that an epoch is a week, and file availability is computed for 2 years (about 100 epochs). Then a 10^{-6} unavailability target for 3 years translates to 10^{-8} unavailability per epoch. This level of availability 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 Figure 3.

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, of course, 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	Detection
1	3	1	$2 \cdot 10^{-6}$	0.999999
1	4	2	$3 \cdot 10^{-6}$	0.999999
1	5	3	$4 \cdot 10^{-6}$	0.999999
1	6	2	$4 \cdot 10^{-9}$	0.99999
2	5	2	$3 \cdot 10^{-6}$	0.999999
2	6	3	$4 \cdot 10^{-6}$	0.999999
2	7	4	$5 \cdot 10^{-6}$	0.999999
2	8	3	$6 \cdot 10^{-9}$	0.99999
3	6	2	$3 \cdot 10^{-6}$	0.999999
3	7	3	$3 \cdot 10^{-6}$	0.999999
3	8	4	$5 \cdot 10^{-6}$	0.999999
3	9	3	$6 \cdot 10^{-9}$	0.99999

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

8 Implementation

We implemented file-encoding functions for HAIL with different parameters for the dispersal code. We performed our experiments using Java on an Intel Core 2 processor running at 2.16 GHz. The JVM was given 1GB of memory and all cryptographic operations use the Java implementation of RSA BSAFE.

For the dispersal code implementation, we used the Jerasure [33] optimized library written in C. The server code in our implementation is constructed using an off-the-shelf Reed-Solomon (223,255,32) encoder over $GF[2^8]$, extended via striping to operate on 32-byte symbols, i.e., $GF[2^{256}]$. To obtain a server code with 4% redundancy, we truncate the last 23 symbols of a codeword, effectively implementing a (223,232,9) server code.

The file is encoded in a single pass to minimize the cost of disk accesses. Since we use Reed-Solomon codes for implementing both the dispersal and server code, computation of parity blocks involves associative operations in a Galois field. For an incremental encoding of the file (i.e., in a single pass), we store the parity blocks into main memory, and update them when we process a file chunk in memory. However, we pay some performance cost for incremental encoding, since existing Reed-Solomon implementations are highly optimized if the whole message is available at encoding time.

Figure 5 shows the encoding cost in HAIL for a 1GB file divided into several components: server code application, dispersal code encoding, and application of PRF to the parity blocks. 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 23. 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.

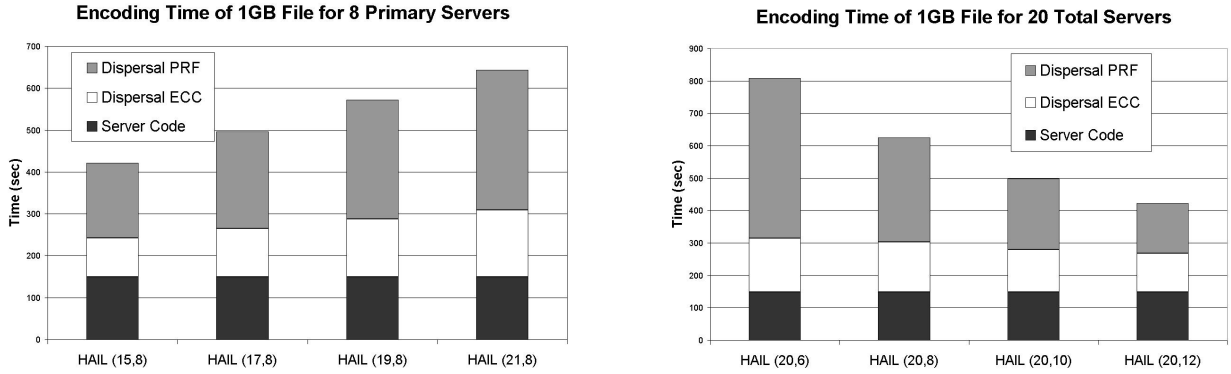

Figure 5. Encoding time for HAIL: on the left, constant number of primary servers (8); on the right, constant total number of servers (20).

As noticed from these graphs, the server code cost is only depending on the file size, and is independent on the number of servers. The cost of the dispersal code increases linearly with the number of secondary servers in the protocol. The high performance cost of PRF application to the parity blocks is mainly due to the Java BSAFE implementation.

The results are shown for our unoptimized Java encoding. We are currently working on a C version of HAIL encoding, which we expect to substantially improve encoding performance.

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. Due to lack of space, we have omitted a variant that efficiently accommodates *file updates*, i.e., small changes to blocks of F. (Efficient file updates are not possible in single-server PORs.) We briefly sketch the idea here. The dispersal code described above stripes blocks according to a fixed structure, namely "horizontally" across servers. Consequently, an adversary that observes modification of a file block on one server can, when moving to other servers, easily locate and attack blocks belonging to the same dispersal codeword. Such an adversary can corrupt a file block in a surgically precise, making such corruption hard to detect.

It is possible, however, to *randomize* the position of dispersal codeword symbols individually on servers. If blocks are then proactively shuffled on each server after every epoch, an adversary can not feasibly infer the position of dispersal codeword elements across servers. It therefore can not corrupt a dispersal codeword without a high level of collateral damage, i.e., a globally detectible level of corruption. By restricting randomization and shuffling to parity blocks, we can render the proactivization step more efficient.

We believe that the HAIL techniques we have introduced in this paper help pave the way for other this and other valuable approaches to distributed file system availability.

Acknowledgements

We thank James Hendricks, Burt Kaliski and Ron Rivest for carefully reading the paper and providing useful comments.

References

[1] Amazon.com. Amazon simple storage service (Amazon S3), 2008. Referenced 2008 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 *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 '99*, pages 216–233. Springer, 1999. LNCS vol. 1666.

[5] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[6] K. Bowers, A. Juels, and A Oprea. Proofs of retrievability: Theory and implementation, 2008. Available from eprint.

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

[8] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *24th Symposium on Reliable Distributed Systems (SRDS 2005)*, pages 191–202, 2005.

[9] L. Carter and M. Wegman. Universal hash functions. *Journal of Computer and System Sciences*, 18(3):143–154, 1979.

[10] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *4th Symposium on Operating System Design and Implementation (OSDI)*, pages 277–283, 2000.

[11] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *4th ACM International Workshop on Storage Security and Survivability (StorageSS)*, 2008.

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

[13] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[14] M. Etzel, S. Patel, and Z. Ramzan. Sqaure hash: Fast message authentication via optimized universal hash functions. In *CRYPTO '99*, pages 234–251. Springer, 1999. LNCS vol. 1666.

[15] D.L.G. Filho and P.S.L.M. Barreto. Demonstrating data possession and uncheatable data transfer, 2006. IACR eArchive 2006/150. Referenced 2008 at http://eprint.iacr.org/2006/150.pdf.

[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] N. Gohring. Amazon's S3 down for several hours, 2008. Available from `http://www.pcworld.com/businesscenter/article/142549/amazons_s3_down_for_several_hours.html`.

[18] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *34th International Conference on Dependable Systems and Networks (DSN 2004)*, pages 135–144, 2004.

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

[20] S. Halevi and H. Krawczyk. Mmh: Software message authentication in the gbit/second rates. In *Fast Software Encryption*, pages 172–189. Springer, 1997. LNCS vol. 1267.

[21] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *26th ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[22] A. Herzberg, M. Jakobsson, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *ACM Computer and Communication Security Conference (CCS)*, 1997.

[23] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or: How to cope with perpetual leakage. In *CRYPTO 1995*, pages 339–352. Springer, 1995. LNCS vol. 963.

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

[25] H. Krawczyk. LFSR-based hashing and authentication. In *CRYPTO 1994*, pages 129–139. Springer, 1994. LNCS vol. 839.

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

[27] R. J. Lipton. A new approach to information theory. In *11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 699–708, 2004.

[28] W. Litwin and T. J. E. Schwarz. Algebraic signatures for scalable distributed data structures. In *20th International Conference on Data Engineering (ICDE)*, 2004.

[29] S. Micali, C. Peikert, M. Sudan, and D. Wilson. Optimal error correction against computationally bounded noise. In *TCC*, pages 1–16. Springer, 2005. LNCS vol. 3378.

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

[31] W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In *Advances in Cryptology – Eurocrypt '97*, pages 24–41. Springer, 1997.

[32] R. Ostrovsky, O. Pandey, and A. Sahai. Private locally decodable codes. In *34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, pages 387–398. Springer, 2007. LNCS vol. 4596.

[33] 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 *USENIX FAST*, 2009.

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

[35] T. J. E. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[36] H. Shacham and B. Waters. Compact proofs of retrievability. In *Asiacrypt*, 2008. To appear. Preprint IACR ePrint manuscript 2008/073.

[37] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest source. In *USENIX HotOS*, page Article Number 11, 2007.

[38] V. Shoup. On fast and provably secure message authentication based on universal hashing. In *CRYPTO '96*, pages 313–328. Springer, 1996. LNCS vol. 1109.

[39] 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.