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. 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 also report on a prototype implementation.

HAIL strengthens, formally unifies, and streamlines distinct approaches from the cryptographic and distributed-systems communities. HAIL also includes an optional new tool for proactive protection of stored files. HAIL is primarily designed to protect static stored objects, such as backup files or archives.

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) and proofs of data possession (PDPs) [2, 23]. 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.[1]

As a standalone tool for testing file retrievability against a single server, though, a POR is of limited value.[2] 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. Surprisingly, the application of PORs to distributed systems has remained unexplored in the literature.

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

In this paper, we unify the two approaches to remote file-integrity assurance in a system that we call *HAIL (High-Availability and Integrity Layer)*. HAIL reaps the benefits from both approaches to achieve the following characteristics:

Intactness proofs: HAIL enables a set of servers to prove

[1]PDPs do not carry the rigorous formal guarantees we wish to establish in HAIL, so we explore only PORs in this paper.

[2]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.

to a client that a stored file F is fully intact—more precisely, that the client can run an interactive extraction algorithm that retrieves 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 contrast, some information dispersal algorithms involve server-to-server protocols, e.g., [16, 18, 9, 20].) The client stores just a secret key.

Static files: HAIL aims to protect static stored objects, such as backup files and archives. It is not generally efficient for highly dynamic objects.

HAIL has the following main technical components:

Three error-correcting codes: As in previous POR constructions, HAIL uses two error-correcting codes within each server, an *inner code* for response computation and *outer code* for file redundancy. HAIL also uses a third error-correcting code called a *dispersal code* across servers—essentially a form of Rabin information dispersal [29].

Homomorphic message-authentication code (MAC): We introduce in HAIL a cryptographic primitive that we call a homomorphic MAC. This tool allows proof values across multiple points in a file to be compressed into a single value.

Reactive reallocation of file shares: HAIL employs a *reactive* strategy that we call test-and-reallocate (TAR). When the client detects a faulty server, it reallocates file shares. (We also describe stronger, *proactive* techniques.)

In more detail, our major technical contributions in HAIL are:

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

None of the existing approaches to client-based file-integrity verification treats the case of a mobile adversary. The POR literature emphasizes primitive-level design and does not explicitly model distributed systems, while the distributed file-system literature usually dispenses with formal adversarial modeling altogether.

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) While designed like a proactive cryptographic system to withstand a mobile adversary, HAIL aims to protect integrity, rather than secrecy. It can therefore be *reactive*. We base HAIL on a new protocol-design strategy that we call TAR (Test-And-Redistribute). With TAR, HAIL relies on PORs to detect file corruption and trigger reallocation of resources when needed—and only when needed. On detecting a fault in a given server, 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, e.g. [25, 31], are largely *underengineered*: Lacking robust testing and reallocation, they provide inadequate protection against mobile adversaries.

As an additional, novel contribution, we propose an optional proactivizing operation that strengthens HAIL at the cost of local scrambling of file contents by servers.

Unified framework for error-coding Previous proposals have relied on a patchwork of error-correcting and error-detection techniques, often fragmentary or disguised versions of earlier constructions. As a first step in our construction of HAIL, we uncover formal technical foundations implicit in previous work, resulting in both unifications and improvements. We create a general framework for file-integrity checking in a construction that we call an *integrity-protected homomorphic ECC* ($\mathsf{IPH\text{-}ECC}$). This construction draws together PRFs, ECCs, and universal hash functions into a single primitive. It is based on three properties of (certain) universal hash functions h, briefly: (1) h is homomorphic, 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 (PRF) g, the function $h_\kappa(m) + g_\kappa(m)$ is a cryptographic message-authentication codes (MAC) on m; and (3) $h_\kappa(m)$

may be treated as a parity block in an error-correcting code applied to m.

Organization We review related work in section 2. We describe technical building blocks for error-correction and error-detection in HAIL in section 3, and present our adversarial model in section 4. We present the HAIL protocol itself in section 5, and give implementation results in section 6. We conclude in section 7.

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, 9, 20]. In these algorithms, file integrity is enforced within the pool of servers itself. Some protocols protect against faulty clients that send inconsistent shares to different servers [18, 9, 20]. 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. HAIL assumes synchronous communication between the client and servers. Unlike previous work, which verifies integrity at the level of individual file blocks, HAIL operates 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 IPH-ECC primitive fuses several threads of research that have emerged independently (and without mutual awareness). At the heart of this research are *Universal Hash-Functions* (UHFs). (In the distributed systems literature, a common term for essentially the same construction is *algebraic fingerprint [26, 31, 20].*) UHFs can be used to construct *message-authentication codes (MAC)* [5, 19, 14], essentially secret-key digital signatures (see [28] for a performance evaluation of various schemes). In particular, a natural combination of UHFs with pseudorandom functions (PRFs) yields MACs that are homomorphic; these MACs can be aggregated over many data blocks and thus support compact proofs over large file samples. One of our contributions in this paper is to show how UHFs and MACs crop up implicitly in several research papers and to extend hitherto unidentified connections in previous work.

PORs and PDPs Juels and Kaliski (JK) [23] propose a formal POR protocol definition and security definitions that we describe below. 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) [32] use an implicit homomorphic MAC construction that enables an unlimited number of queries, at the expense of larger storage overhead. Their homomorphic MAC construction is based on the UHF + PRF paradigm, but they seem unaware of their implicit use of a UHF. They construct a UHF based on a random linear function, rather than a more efficient, standard error-correcting code.

Bowers et al. (BJO) [7] give a general framework for POR protocols that generalizes and improves both the JK and SW protocols. We adopt elements of their framework in the construction of the HAIL protocol.

Ateniese et al. [2] propose a closely related construction called a *proof of data possession* (PDP). A PDP demonstrates to a client that a server possesses a file F (in an informal sense). It is weaker than a POR in not guaranteeing that the client can retrieve F. The lack of error correction and extraction algorithms in basic PDP constructions permit file updates to be performed efficiently [3]. Curtmola et al. [13] 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 an impractical PDP scheme based on full-file processing. Shah et al. [33] consider a symmetric-key variant, but their scheme only works for encrypted files, and auditors must maintain long-term state. Naor and Rothblum [27], extending the memory-checking schemes of Blum et al. [6], describe a theoretical model that may be viewed as a generalization of PORs. However, NR do not naturally model PORs with non-trivial challenge-response protocols, as required for our purposes in this paper.

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

Schwartz and Miller (SM) [31] propose a scheme that ensures file integrity through distribution across multiple servers, using error-correcting codes and block-level file integrity checks. They employ keyed algebraic encoding and stream-cipher encryption to detect file corruptions. While they do not explicitly recognize the fact, 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 [35]. (In general, they propose ad-hoc but clever techniques.) We adopt this tool of simultaneous MACing and error-correcting in our HAIL constructions.

Neither Lillibridge et al. nor SM gives a formal adversarial model or rigorous security analysis of their protocols.

Proactive cryptography Our adversarial model is inspired by the literature on proactive cryptography initiated by [22], which has yielded protocols resilent to mobile adversaries for secret sharing [22, 8] as well as signature schemes [21].

Proactive recovery has been proposed for the BFT system by Castro and Liskov [11]. Their system constructs a replicated state machine that tolerates a third of faulty replica 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 Building Blocks

3.1 UHFs

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

A UHF [10] is an algebraic function $h : \mathcal{K} \times I^k \rightarrow I$ that compresses a message or file element $m \in I^k$ 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 and 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 $x \neq y \in I^k$: $Pr_{\kappa \leftarrow \mathcal{K}}[h_\kappa(x) = h_\kappa(y)] \leq \epsilon$.*

h is an ϵ-AXU family if for any $x \neq y \in I^k$, 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).

For example, as we assume for convenience in our work here, a UHF may be based on a generalized (n, k)-Reed-Solomon code over I. Let $\vec{m} = (m_1, m_2, \ldots, m_k)$, where $m_i \in I$. In this case, recall that an R-S code may be viewed in terms of a polynomial representation of m of the form $p_{\vec{m}}(x) = m_k x^{k-1} + m_{k-1} x^{k-2} + \ldots + m_1$. A generalized R-S codeword, then, may be defined in terms of a vector $\vec{x} = (x_1, \ldots, x_n)$: The corresponding codeword is $(p_{\vec{m}}(x_1), p_{\vec{m}}(x_2), \ldots, p_{\vec{m}}(x_n))$.

A UHF of interest, then, is simply $h_\kappa(m) = p_{\vec{m}}(\kappa)$, the κ^{th} symbol of the Reed-Solomon codeword for m. Here $\mathcal{K} = I$. It is well known that this construction, which we refer to as RS-UHF, is indeed a good UHF [34]:

Fact 1 RS-UHF *is a $\frac{k-1}{2^l}$-AXU hash family.*

3.2 Homomorphic MACs: The UHF + PRF construction

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 MAC is given by three algorithms: $\kappa \leftarrow$ MGen generates a secret key given a security parameter; $\tau \leftarrow \mathsf{MAC}_\kappa(m)$ 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* MAC *and* MVer *oracles, whose goal is to output a valid MAC on a message not queried to* MAC. *We define:*

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

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

It is well known that a MAC may be constructed as the straightforward composition of a UHF with a *pseudorandom function* (PRF) [35, 24, 30, 34]. A PRF is a keyed family of functions $g : \mathcal{K}_{\mathsf{PRF}} \times D \rightarrow R$ that 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 $\mathsf{Adv}_g^{\mathsf{prf}}(\mathcal{A}) = |\Pr[\kappa \leftarrow \mathcal{K}_{\mathsf{PRF}} : \mathcal{A}^{g_\kappa} = 1] - \Pr[z \leftarrow \mathcal{F}^{D \rightarrow R} : \mathcal{A}^z = 1]|$, where $\mathcal{F}$ is the set of all functions from D to R. We denote by $\mathsf{Adv}_g^{\mathsf{prf}}(q,t)$ the maximum prf-advantage of an adversary making q queries to its oracle and running in time t.

Given a UHF family $h : \mathcal{K} \times I^k \rightarrow I$ and a PRF family $g : \mathcal{K}' \times L \rightarrow I$, the function $f : \mathcal{K} \times \mathcal{K}' \times I^k \times L \rightarrow I$ defined as $f_{\kappa,\kappa'}(m,r) = h_\kappa(m) + g_{\kappa'}(r)$, with key (κ,κ') generated uniformly at random from $\mathcal{K} \times \mathcal{K}'$, and r a unique message tag or label from space L, is a MAC, whose properties are as follows [34].

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

Given a linear UHF, an interesting feature of the MAC f is that it is *homomorphic*. Let $\vec{M} = (m_1,\ldots,m_v) \in I^v$ be a set of messages and let $\vec{A} = (\alpha_1,\ldots,\alpha_v) \in I^v$ be a set of scalar values. We define $\tau = \sum_{i=1}^v \alpha_i f(m_i,r_i)$ as the *composite* MAC of $\vec{M}$ for coefficients $\alpha_1,\ldots,\alpha_v$. If τ is the composite MAC of $\{m_i,r_i\}_{i=1}^v$ for coefficients $\{\alpha_i\}_{i=1}^v$, the composite MAC verification algorithm $\mathsf{CMVer}(\{m_i,r_i,\alpha_i\}_{i=1}^v,\tau)$ outputs 1. Consider an adversary that has access to MAC and CMVer oracles. Intuitively, a homomorphic MAC is such that the adversary can generate a vector of messages and a composite MAC with negligible probability if it does not query the MAC oracle for *all* component messages of the vector.

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

Definition 3 *Let (*MGen*,*MAC*,*MVer*) be a MAC algorithm. Consider an adversary $\mathcal{A}$ with access to MAC and* CMVer *oracles whose goal is to output a set of messages $m_1,\ldots,m_v$ with tags $r_1,\ldots,r_v$, a set of coefficients $\alpha_1,\ldots,\alpha_v$ and a composite MAC τ. We define:*

$\mathsf{Adv}_{\mathsf{MAC}}^{\mathsf{h\text{-}mac}}(\mathcal{A}) = \Pr[\kappa \leftarrow \mathsf{MGen}; (\{m_i,r_i,\alpha_i\}_{i=1}^v,\tau) \leftarrow \mathcal{A}^{\mathsf{MAC}_\kappa,\mathsf{CMVer}} : \mathsf{CMVer}(\{m_i,r_i,\alpha_i\}_{i=1}^v,\tau) = 1 \wedge \exists i \in [1,v]$ *for which m_i was not queried to* $\mathsf{MAC}_\kappa]$.

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

Lemma 1 *The homomorphic MAC f defined above has advantage:* $\mathsf{Adv}_f^{\mathsf{h\text{-}mac}}(q_1^h,q_2^h,t_h) \leq \mathsf{Adv}_f^{\mathsf{uf\text{-}mac}}(q_1^h+vq_2^h+v-1,0,(v+1)t_h)$.

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

We build an adversary $\mathcal{A}'$ that targets the MAC. $\mathcal{A}'$ runs $\mathcal{A}$. For any query that $\mathcal{A}$ makes to the MAC oracle, $\mathcal{A}'$ returns the output of its own MAC oracle. For any query that $\mathcal{A}$ makes to the composite MAC verification oracle, $\mathcal{A}'$ queries its MAC oracle v times, and checks the linear relation.

When $\mathcal{A}$ 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, $\mathcal{A}'$ queries the MAC oracle for $m_1,\ldots,m_{v-1}$ and then outputs m_v and its MAC computed as $f(m_v,r_v) = \alpha_v^{-1}(\tau - \sum_{j=1}^{v-1}\alpha_j f(m_j,r_j))$.

$\mathcal{A}'$ makes $q_1 = q_1^h + vq_2^h + (v-1)$ queries to the MAC oracle and no queries to the verification oracle. Its running time is at most $v+1$ times the running time of $\mathcal{A}$. The lemma follows. ▮

Schwartz and Miller propose an ad-hoc construction that appeals implicitly to this lemma. (They appear to be unaware, however, of the UHF + PRF construction, and do not specify formal properties.) Similarly, Shacham and Waters use a homomorphic MAC construction of essentially this type. They, however, appear to be unaware of their implicit use of a UHF. They construct a UHF based on a random linear function, rather than benefitting from the efficiencies of a canonical error-correcting code.

3.3 An integrity-protected, homomorphic error-correcting code (IPH-ECC)

The homomorphic MAC construction we have outlined presumes the ordinary cryptographic mode of use in which a MAC is appended to a message. A more general view is possible in which the message is transformed into a codeword and the output of a PRF is applied to an arbitrary subset of symbols (and possibly all of them). In this more general view, knowledge of (κ,κ') permits message recovery with the full properties of the underlying error-correcting code. This is particularly useful in a distributed setting. Intuitively, thanks to use of a PRF, it is possible to "piggyback" on the redundancy across servers to construct a MAC with no additional storage costs. Schwartz and Miller adopt essentially this approach (again, without the knowledge that they are implementing a homomorphic MAC scheme). We define an integrity-protected, homomorphic ECC (IPH-ECC) as follows:

Definition 4 *An (n,k,d)* integrity-protected homomorphic ECC *is defined as an error-correcting code that encodes messages of k symbols into codewords of n symbols using a secret key κ and has minimum distance d. It has the additional property that the codeword $(c_1,\ldots,c_n)$ is a homomorphic MAC on the message $(m_1,\ldots,m_k)$. (A similar definition is possible for erasure codes.)*

Our IPH-ECC construction is a function H in which a generalized Reed-Solomon code is applied to a message $M = (m_1, m_2, \ldots, m_k)$, followed by the application of a PRF function g. In particular, let $\vec{\kappa} = (\kappa_1, \ldots, \kappa_n)$ and likewise for $\vec{\kappa'}$. We define the encoding of message M and a unique message "tag" (the file handle concatenated with the position of the message in a file), as $H_{\vec{\kappa},\vec{\kappa'}}(M, \tau) = (c_1, \ldots, c_n)$, where $c_i = \mathsf{RS\text{-}UHF}_{\kappa_i}(M) + g_{\kappa'_i}(\tau), i = [1, n]$.

For a systematic version of H, we apply the PRF function only to the parity blocks. We define $c_i = m_i, i = [1, k]$ and $c_i = \mathsf{RS\text{-}UHF}_{\kappa_i}(M) + g_{\kappa'_i}(\tau), i = [k+1, n]$. We employ this systematic IPH-ECC construction for our protocols in this paper.

Lemma 2 *If* RS-UHF *is constructed from a* (n, k, d)*-Reed-Solomon code and g is a PRF family, then the code H defined above is a* (n, k, d)*-integrity-protected homomorphic code, with advantage*

$$\mathsf{Adv}_H^{\text{h-mac}}(q_1, q_2, t) \leq \left[\mathsf{Adv}_f^{\text{h-mac}}(q_1, q_2, t)\right]^{n-k}.$$

Proof: H is an error correcting code (or erasure code) with the same (n, k, d) parameters as the underlying Reed-Solomon code. The decoding algorithm can be built straightforward by stripping off the PRF from the codeword symbols, and then decoding under the Reed-Solomon code the resulting codeword symbols.

Let $\mathcal{A}$ be an adversary algorithm for H that makes q_1 queries to the MAC oracle, q_2 queries to the composite MAC verification oracle and runs in time t. It outputs a message $(m_1, \ldots, m_k)$ and its encoding $(c_1, \ldots, c_n)$ that satisfies $n-k$ linear equations in I (to be a valid codeword in the Reed-Solomon code). We can then build $n-k$ adversaries for the homomorphic MAC f, one for each linear relation (i.e., the composite MAC verification oracle for each adversary is the corresponding linear relation). Each such adversary runs $\mathcal{A}$, replies to its queries using its own oracles, and outputs what $\mathcal{A}$ outputs. If $\mathcal{A}$ replies with a valid codeword, all constructed adversaries succeed in breaking the homomorphic MAC f. ▌

3.4 Adversarial codes

Adversarial ECCs [7, 23] 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 resilence against a computationally bounded adversary.

An (n, k, d)-error-correcting code corrects up to $\frac{d}{2}$ errors, and thus it supports a fraction of $\frac{d}{2n}$ adversarial corruptions. 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 [7] first define adversarial codes formally and give a 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. [12].

BJO define an adversarial ECC as follows:

Definition 5 *An* (n, k, d)-adversarial error-correcting code AECC *consists of a private key space* SK*, an alphabet* Σ*, and a triple of functions: (1) a probabilistic function* KGenECC : $1^l \rightarrow \kappa \in SK$*; (2) a deterministic function* enc : $SK \times \Sigma^k \rightarrow \Sigma^n$*; and (3) a deterministic function* dec : $SK \times \Sigma^n \rightarrow \Sigma^k$.

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

$\mathsf{Adv}_{\mathsf{AECC}}^{\text{adv-ecc}}(\mathcal{A}, \epsilon, \delta) = \Pr[\kappa \leftarrow SK; (c, c') \leftarrow \mathcal{A}() : (c \neq c') \wedge (|c - c'| \leq \epsilon n)] - \delta$.

A $(\epsilon, \gamma, \beta)$*-adversarial code is such that* $\mathsf{Adv}_{\mathsf{AECC}}^{\text{adv-ecc}}(\mathcal{A}, \epsilon, \delta)$ *is bounded above by* β*, for all polynomial-time adversaries* $\mathcal{A}$*.*

The definition can be easily extended to erasure codes It is easy to show that an (n, k, d)-ECC that is a maximum-distance separable (MDS) $(\epsilon, \gamma, \beta)$-adversarial code with $\epsilon \leq \frac{d}{2n}$, is also a $(2\epsilon, \gamma, \beta)$-adversarial erasure code.

4 Adversarial Model

We model a distributed file system for 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. The adversary $\mathcal{A}$ is *mobile*, i.e., can corrupt a different set of servers in each timestep. In practice $\mathcal{T}$ may be a client or an external auditor. We assume that there exists known upper bounds on message delivery in the network, and as such our system operates in a synchronous communication model.

A time step consists of three phases:

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

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

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 $\mathcal{T}$. Let $F_t^{(i)}$ denote the file share held by server S_i at the beginning of timestep 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. (Of course, $F_{t+1}^{(i)}$ reflects the results of remediation in timestep t.)

Our model can characterize Byzantine adversaries $\mathcal{A}$ that can corrupt $F_t^{(i)}$ arbitrarily. More limited adversaries may also be modelled. For instance, for *erasing* adversary, $\hat{F}_t^{(i)}$ is a subset of $F_t^{(i)}$.

After a server undergoes remediation, we assume that it then operates honestly, i.e., presents a correct application-level interface, until it is again corrupted. This assumption is necessary for a meaningful security model. As every server may ultimately be corrupted at some time, permanent software corruption would allow the adversary to cause all servers to fail simultaneously. For example, $\mathcal{A}$ might program all servers to fail at a pre-determined time or collectively refuse to respond when the client seeks to retrieve F.

4.1 POR

Following [23], a POR comprises six functions. We present modified versions of these functions for the distributed setting, and refer to them collectively as a HAIL system. For clarity, we assume that the client is stateful and omit the state variable ω. We also omit file handle η and leave the system parameters π implicit where clearer (including key lengths, etc.). A system HAIL comprises the following POR functions:

- $\mathsf{keygen} \rightarrow \kappa$: The function keygen generates a key $\kappa = (sk, pk)$. (For symmetric-key PORs, pk may be null.)
- $\mathsf{encode}(F; k, n, b, \kappa) \rightarrow \{F_0^{(i)}\}_{i=1}^n$: 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 k-out-of-n redundancy across servers and to provide resilience against an adversary that can corrupt at most b servers in any time step.
- $\mathsf{challenge}(\kappa) \rightarrow \{c_i\}_{i=1}^n$: The function challenge generates a challenge value c_i for each server i.
- $\mathsf{respond}(i, c) \rightarrow r$: The function respond generates response r from server i to challenge c.
- $\mathsf{verify}(\{c_i, r_i\}_{i=1}^n; \kappa) \rightarrow b \in \{0, 1\}$. The function verify checks whether $r_1, \ldots, r_n$ is a valid response to challenge $c_1, \ldots, c_n$. 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.
- $\mathsf{extract}(\kappa) \rightarrow F'$: The function extract exploits the challenge-response interface of the POR in the HAIL system to recover the file. It dynamically determines a sequence of challenges that the client sends to each server. If successful, $F' = F$; otherwise, we assume that $F' = \perp$.

To extend the POR definition for HAIL, we require an additional function:
- $\mathsf{redistribute}(\kappa)$: The function redistribute is an interactive protocol that replaces $\{\hat{F}_j^{(i)}\}_{i=1}^n$ with $\{F_{j+1}^{(i)}\}_{i=1}^n$. It implements a recreation and distribution of segments of F. We leave the definition of redistribute as general as possible. The function may involve the client extracting segments, reconstructing F, and reinvoking encode.

We also consider an optional function:
- $\mathsf{proactivize}(\kappa_t^{(i)}, F_t^{(i)}) \rightarrow (\kappa_{t+1}^{'(i)}, F_{t+1}^{'(i)})$: A server i executes the function proactivize using server-specific key $\kappa_t^{(i)}$ to reorder a subset of file blocks in segment $F_t^{(i)}$, typically through file-block permutation. The function updates both $F^{(i)}$ and the key.

More general definitions for HAIL are possible, but excluded here. For example challenge and respond might be defined as multi-server interactive protocols.

4.2 Security model

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 $(k, n, b, T, \epsilon_c, n_c)$.

$\mathcal{A}$ participates in the two-phase experiment in Figure 1. 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 shares of at most b out of the n servers. Each server is challenged n_c times in each interval, and $\mathcal{A}$ responds to the challenges sent to the corrupted servers. If more than a fraction ϵ_c of its responses are incorrect, the redistribute algorithm is invoked.

$\mathcal{A}$ is successful if the experiment outputs 1, i.e., the file can not be correctly extracted. 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 $\mathsf{Adv}^{\mathsf{HAIL}}(\pi, q_1, l_e, q_2, t)$ the maximum advantage of all adversaries making q_1 queries to encode of total length l_e, q_2 queries to verify, and running in time at most t.

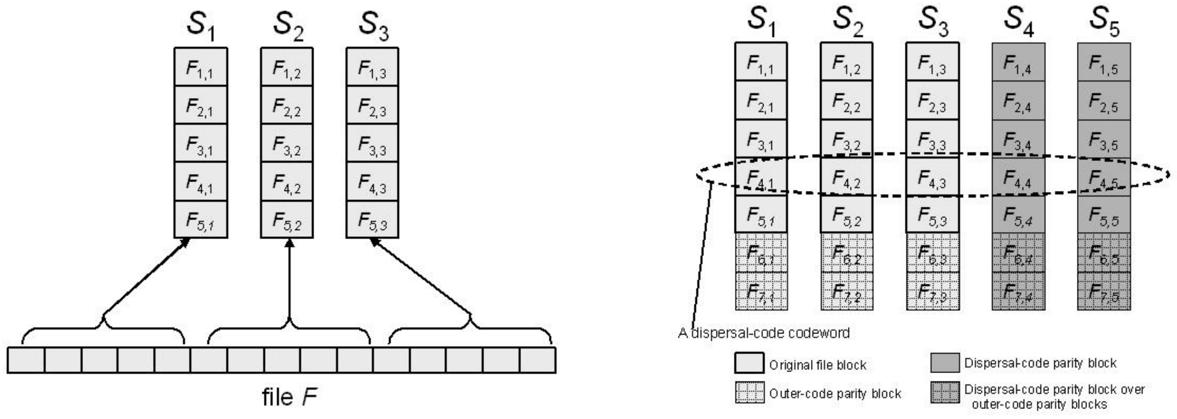
S_1
S_2
S_3
$F_{1,1}$ $F_{2,1}$ $F_{3,1}$ $F_{4,1}$ $F_{5,1}$
$F_{1,2}$ $F_{2,2}$ $F_{3,2}$ $F_{4,2}$ $F_{5,2}$
$F_{1,3}$ $F_{2,3}$ $F_{3,3}$ $F_{4,3}$ $F_{5,3}$
file F
S_1
S_2
S_3
S_4
S_5
$F_{1,4}$ $F_{2,4}$ $F_{3,4}$ $F_{4,4}$ $F_{5,4}$ $F_{6,4}$ $F_{7,4}$
$F_{1,5}$ $F_{2,5}$ $F_{3,5}$ $F_{4,5}$ $F_{5,5}$ $F_{6,5}$ $F_{7,5}$
$F_{6,1}$ $F_{7,1}$ $F_{6,2}$ $F_{7,2}$ $F_{6,3}$ $F_{7,3}$
A dispersal-code codeword
Original file block
Outer-code parity block
Dispersal-code parity block
Dispersal-code parity block over outer-code parity blocks

Figure 2. 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 outer and dispersal codes.

```
Experiment $\mathbf{Exp}^{\mathsf{HAIL}}_{\mathcal{A}}(\pi)$:
  $\kappa = (sk, pk) \leftarrow \mathsf{keygen}$
  $F \leftarrow \mathcal{A}(\text{“test”}, pk, \pi)$
  $\{F_0^{(i)}\}_{i=1}^n \leftarrow \mathsf{encode}(F; k, n, b, \kappa)$
  for $j = 0$ to $T$ do
    $C_j \leftarrow \mathcal{A}(\text{“corrupt servers”}, j)$
    $\{\hat{F}_j^{(i)}\}_{i \in C_j} \leftarrow \mathcal{A}(\text{“modify segments”}, \{F_j^{(i)}\}_{i \in C_j})$
    $V \leftarrow 0$
    for $l = 1$ to $n_c$ do
      $c = (c_1, \ldots, c_n) \leftarrow \mathsf{challenge}(\kappa)$
      for $i = 1$ to $n$ do
        if $i \in C_j$ then $r_i \leftarrow \mathcal{A}(\text{“respond”}, c_i)$
        else $r_i \leftarrow \mathsf{respond}(i, c_i)$
      if $\mathsf{verify}(\{c_i, r_i\}_{i=1}^n; \kappa) = 1$ then
        $V \leftarrow V + 1$
    if $\frac{V}{n_c} < 1 - \epsilon_c$ then redistribute
    else $\{F_{j+1}^{(i)}\}_{i=1}^n \leftarrow \{\hat{F}_j^{(i)}\}_{i=1}^n$
  if $\mathsf{extract}(\kappa) = F$ then output ‘0’
  else output ‘1’
```

Figure 1. HAIL security experiment.

5 HAIL: Protocol Specification

In this section, we give full details on the HAIL protocol. We first review the key pieces of intuition.

In HAIL, the file F is distributed such that each server S_j holds a fragment $F^{(j)}$ (whose exact form we specify below). This fragment $F^{(j)}$ is encoded under a code $\mathsf{ECC_{out}}$ referred to in [7] as the *outer code*. The effect of the outer code $\mathsf{ECC_{out}}$ is to render $F^{(j)}$ robust to corruptions within S_j. With a good choice of outer code, $F^{(j)}$ can be made robust to multiple errors. For example, in a practical parameterization of our system, we might choose a code $\mathsf{ECC_{out}}$ that tolerates corruption of up to 10% of the blocks in $F^{(j)}$.

Given the use of a good outer code, then, an adversary must corrupt a substantial fraction of $F^{(j)}$ to render it irretrievable. Consequently, by performing random integrity checks on the blocks of $F^{(j)}$, the client can detect irretrievable corruption of $F^{(j)}$ with high probability. (In contrast, without encoding, a single bit corruption could be fatal, but nearly undetectable.)

While the client could check individual blocks in $F^{(j)}$, a more efficient approach is to check multiple blocks of $F^{(j)}$ *simultaneously*. This is where the homomorphic MAC construction has its place. The client can specify multiple positions in $F^{(j)}$, and verify their correctness via a single, composite response from S_j. (For extra bandwidth savings, the client can specify positions implicitly using a pseudorandom seed.)

An important guarantee provided by the use of a POR in HAIL is that the fragment $F^{(j)}$ can be *extracted*: The client can retrieve $F^{(j)}$ with overwhelming probability using the challenge-response interface. To ensure extraction, the homomorphically composed responses in the challenge-response protocol in HAIL assume a special form. The responses are computed on the fly as symbols in a second error-correcting code known as the *inner code*. (While technically important, the inner code isn't essential to an intuitive understanding of HAIL.)

How does the client verify the correctness of the homomorphic MAC responses made by S_j? In the single-server POR constructions of prior work, check values on $F^{(j)}$ are explicitly stored on S_j along with $F^{(j)}$ itself. For instance, in the Shacham-Waters construction, homomorphic MACs are stored on segments of $F^{(j)}$. In contrast, in HAIL, *there are no explicit check values within servers.* Instead, file redundancy *across servers* provides the check values needed to verify server responses within the POR framework.

This redundancy across servers is achieved via a third error-correcting code that we call a *dispersal code* and denote by $\mathsf{ECC}_{\mathsf{disp}}$. The dispersal code is an (n, k_{disp})-error correcting code that distributes the file F across n servers. Natural choices of $\mathsf{ECC}_{\mathsf{disp}}$ are systematic ones, i.e., codes that partition F across k_{disp} servers and place parity blocks on the remaining $n - k_{\mathsf{disp}}$ servers. A key feature of HAIL is the choice of a dispersal code that contains an embedded homomorphic MAC, namely an integrity-protected-homomorphic code IPH-ECC. As explained above, the combination of cryptography and error correction in our integrity-protected-homomorphic code IPH-ECC imposes no more storage costs than error-correction alone; the only added cost is that of PRF computation.

Following our reactive design strategy TAR in HAIL, file shares are recovered and redistributed only when detecting faults through the challenge-response mechanism. Upon detecting a fault, the client downloads all file fragments, decodes the file, and redistributes only the corrupted shares, while leaving unchanged the correct shares.

Notation Let $(k_{\mathsf{disp}}, n, d_{\mathsf{disp}})$, $(k_{\mathsf{out}}, n_{\mathsf{out}}, d_{\mathsf{out}})$ and $(k_{\mathsf{in}}, n_{\mathsf{in}}, d_{\mathsf{in}})$ be the parameters of the dispersal, outer and inner codes, respectively. We assume that all the codes use symbols derived from the same alphabet (the Galois field $I = GF[2^l]$), and thus have the same symbol size $|I| = 2^l$. We require that the codeword size of the dispersal code equal the number of servers n and the number of faults tolerated in each round b be at most d_{disp} .

5.1 Encoding Files

A graphical representation of our encoding algorithm is given in Figure 2. Before transforming the file F into a distributed, encoded representation F_d, we partition F into k_{disp} distinct segments $F^{(1)}, \ldots, F^{(k_{\mathsf{disp}})}$ and distribute these segments across servers $S_1, \ldots, S_{k_{\mathsf{disp}}}$ respectively. This distributed cleartext representation of the file remains untouched by our subsequent encoding steps.

We then encode each segment $F^{(j)}$ under the outer code. The effect of the outer code is to extend the "columns" of F_d by adding parity blocks. Next, we apply the dispersal code. The dispersal code creates the parity blocks that reside on servers $S_{k_{\mathsf{disp}}+1}, \ldots, S_n$. It extends the "rows" of F_d across the full set of n servers. (The dispersal code and outer code in fact commute, so we can swap these two encoding steps, if desired.) To embed the dispersal code in a full-blown IPH-ECC, we also add PRF values on the parity blocks for each row, i.e., on the blocks contained in servers $S_{k_{\mathsf{disp}}+1}, \ldots, S_n$. Viewed another way, we "encrypt" columns $k_{\mathsf{disp}}+1$ 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 client. This full-file MAC allows the client to confirm when it has successfully downloaded the file F.

Recall that inner code values are computed on the fly during the challenge-response protocol. The inner code plays no role in the initial file encoding. The steps of encode are detailed below:

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

2. [Outer code application] Encode each file segment $F^{(j)}$ under the outer systematic code $\mathsf{ECC}_{\mathsf{out}}$, and obtain a segment of m blocks at each server (where blocks $m_F + 1, \ldots, m$ are parity blocks for the outer code).

3. [Dispersal code / IPH-ECC application] Apply the dispersal code $\mathsf{ECC}_{\mathsf{disp}}$ constructed from the IPH-ECC code to the rows of the encoded matrix obtained in step 2. We determine thus the segments $F^{(k_{\mathsf{disp}}+1)}, \ldots, F^{(n)}$.

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 $i = [1, m_F]$, and $F^d_{ij} = \mathsf{RS\text{-}UHF}_{\kappa_j^{\mathsf{disp}}}(F_{i1} \ldots F_{ik_{\mathsf{disp}}}) + g_{\kappa_j^{PRF}}(\tau_{ij})$, for $i = [m_F + 1, m]$. κ_j^{disp} and κ_j^{PRF} are secret keys for selecting the dispersal code symbol and for keying the PRF, respectively. 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., 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 $F_0^{(j)} = \{F^d_{ij}\}_{i=[1,m]}$.

5.2 The Challenge-Response Proofs

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 the set D, as well as a challenge key u. Each server S_j returns a homomorphic aggregate (linear combination) of the blocks in the row positions of D. We denote this aggregate value by R_j.

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 $R_{k_{\mathsf{disp}}+1}, \ldots, R_n$ of the parity servers $S_{k_{\mathsf{disp}}+1}, \ldots, S_n$. Thanks to our IPH-ECC, the client can check the validity of the response R by stripping away the PRF pads and verifying that the result is a valid codeword in the dispersal code.

Other details are exactly as in the POR scheme in [7]. In particular, to ensure that the challenge-response interface can be used for file extraction, each response R_j is computed as the u^{th} symbol in the inner code $\mathsf{ECC}_{\mathsf{in}}$. And for bandwidth efficiency, the client can, of course, specify D as pseudorandom subset computed from a seed s. The client only transmits s then, not D.

To give more details, a challenge c consists of a seed κ_c, from which $v = k_{\mathsf{in}}$ positions $i_1, \ldots, i_v$, as well as an index u are derived. Each server responds with the u-th symbol in the codeword obtained from the encoding of this message under $\mathsf{ECC}_{\mathsf{in}}$. The response to challenge c from server j is $R_j = p_{F^d_{i_1 j} \ldots F^d_{i_v j}}(u)$.

To verify the responses received from the servers, the client applies the verification algorithm of the integrity-protected homomorphic MAC construction. More specifically, the client first strips off the PRF from the responses. This can be done by the client with its PRF secret key:

$$R'_j = R_j - \frac{u^s - 1}{u - 1} \sum_{i=1}^{m} g_{k_j^{PRF}}(\tau_{ij}).$$

In order to check that $(R'_1, \ldots, R'_n)$ is a codeword of $\mathsf{ECC}_{\mathsf{disp}}$, the client needs to check $n - k_{\mathsf{disp}}$ equations:

$$\sum_{j=1}^{n} R'_j \frac{(\kappa_j^{\mathsf{disp}})^{i-1}}{\prod_{l \neq j} (\kappa_j^{\mathsf{disp}} - \kappa_l^{\mathsf{disp}})} = 0, 1 \leq i \leq n - k_{\mathsf{disp}}.$$

5.3 Decoding Files

To recover the original file from the encoded matrix F_d, two layers of decoding (of the dispersal and outer codes) are applied.

Decode from the dispersal code. The first decoding step for the client is to check the MAC implicit in each row of the encoded matrix. If a row is corrupted, there are several options for reconstructing it, depending on the choice of dispersal code:

- If the dispersal code is an error-correcting code, then $\frac{d_{\mathsf{disp}}}{2}$ errors can be corrected in each row. This choice imposes the requirement on $\mathcal{A}$ that $b < \frac{d_{\mathsf{disp}}}{2}$.
- If the dispersal code is an erasure code, then it can correct d_{disp} erasures. In this case, a mechanism for converting the erasure code into an error-correcting code is needed. We can find erroneous blocks using the embedded MAC. This approach requires brute force: We try erasing each set of d_{disp} blocks, recover the message from the other $n - d_{\mathsf{disp}}$ blocks using the dispersal code, and accept if the result matches the embedded MAC.

Using an erasure code instead of an error-correcting code for $\mathsf{ECC}_{\mathsf{disp}}$ requires fewer servers. The required brute-force decoding, though, is asymptotically inefficient, since $\binom{n}{d_{\mathsf{disp}}}$ combinations of blocks have to be examined. We recommend use of an erasure code for $\mathsf{ECC}_{\mathsf{disp}}$ for small values of n (e.g., $n < 10$), and a full-blown error-correcting code for larger values of n.

Decode from the outer code. The blocks in the rows that cannot be reconstructed from the dispersal code are marked as erasures in the matrix. The outer code is then applied to recover those erasures.

5.4 Redistribution of Shares

When a server corruption is detected through the challenge-response mechanism at time t, the client contacts all servers to download their current shares and build matrix F_d. The client applies the decoding algorithm to decode the file and recover the corrupted shares. 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 reply correctly can remain unchanged for time $t + 1$.

5.5 Extraction

Since our encoding of files is systematic, the client could recover the file from the first k_{disp} servers, assuming that they return correct shares. If this first process is unsuccessful, the client downloads the shares from all servers, applies the decoding algorithm, and verifies that the reconstructed file matches the integrity MAC stored locally. If this fails as well, then the client invokes the extract algorithm that uses the challenge-response mechanism provided above. We omit here the full details of the extract algorithm, as it is a straightforward generalization of the algorithm given in [7] for the single-server case.

5.6 Optional Proactivization of Shares

To strengthen the security of our HAIL protocol, it is possible periodically to invoke a proactivize algorithm. This algorithm hides the component blocks of rows by independently permuting file blocks within each server. This permutation operation is keyed using a secret key shared between the server and client, so that the client remains aware of the current ordering of each server's file blocks. In systematic file encoding such as we consider here, proactivize permutes only the parity blocks of the outer code in each server (i.e., rows between $m_F + 1$ and m in the encoded matrix). To achive proactive security, a new shared permutation key must be generated on each invocation of proactivize in a forward-secure manner [4].

5.7 Security Analysis

In our security analysis, we restrict ourselves to the case in which the shares at each server are not proactivized. We assume use of the Reed-Solomon-based adversarial outer code $\mathsf{ECC}_{\mathsf{out}}$ proposed in BJO—but employed for efficiency as an erasure code. The dispersal code $\mathsf{ECC}_{\mathsf{disp}}$ is an integrity-protected, homomorphic code (IPH-ECC as defined in Section 3.3). Our goal is to bound the advantage of the adversary in experiment $\mathbf{Exp}^{\mathsf{HAIL}}$. Here is a roadmap of our proof strategy:

1. [Filter correct replies] The homomorphic MACs in the dispersal code enable us to check the replies received from servers in the challenge-response protocol. Given the properties of homomorphic MACs, we can bound the probability that the adversary is able to forge a correct reply.

2. [Bound fraction of correct replies] Drawing on the algorithmic abstraction of BJO, the challenge-response protocol performed in each interval is used to ensure that a server adversary replies correctly to a fraction $1-\epsilon_c$ of challenges. By specifying enough challenges n_c in each time interval, we can obtain the desired ϵ_c with overwhelming probability.

3. [Bound fraction of corrupted blocks at each server] Drawing on the proofs of BJO, we are able to specify an optimal adversarial strategy, i.e., one that achieves the maximum possible advantage in $\mathbf{Exp}^{\mathsf{HAIL}}$. This adversary marks a fixed set of file blocks on a server as red; all other blocks are colored blue. If a challenge "touches," i.e., includes a red block, then the adversary provides a corrupted response; otherwise, the adversary provides a correct response. We refer to this adversary as a *red-blue adversary*.

Given a particular inner code and challenge-set size, and a red-blue adversary that corrupts at most ϵ_c-fraction of responses, we are able to compute a bound ϵ_b on the fraction of red blocks that the adversary creates on each server. The bounds we prove on the red-blue adversary apply to *any* adversary.

4. [Extraction with inner code] By specifying enough challenges in our extract algorithm—and making appropriate use of our inner code—we can filter out all red blocks and retrieve all blue blocks correctly w.o.p. Given this filtering abstraction provided by techniques in BJO, we can treat the extract algorithm in HAIL as simply obtaining a $(1-\epsilon_b)$-fraction of correct blocks from the original file.

5. [Extraction with adversarial code] File encoding in HAIL obtained as a composition of the outer and dispersal codes is, in fact, an adversarial code, whose exact parameters we determine below. Given a bound on the number of correct blocks extracted from each server, and the parameters of the new adversarial code that, intuitively, randomizes the positions of red blocks, we can effectively compute the probability with which we are able to recover the missing blocks, and thus defeat the adversary in experiment $\mathbf{Exp}^{\mathsf{HAIL}}$.

The MACs in our scheme come into play in two places. First, these MACs serve to verify server responses in the classification phase of HAIL. Second, the MACs permit the extract algorithm to filter out corrupted responses when it uses the challenge-response interface to retrieve file blocks. This filtering process effectively reduces a corrupting adversary to an erasing one. It allows us to use erasure codes as our inner code and outer codes, rather than error-correcting codes.

For filtering correct replies (i.e., step 1 in our outline), we give the following lemma:

Lemma 3 *Consider an adversary $\mathcal{A}$ participating in $\mathbf{Exp}^{\mathsf{HAIL}}_{\mathcal{A}}(\pi)$ making q_1 queries to the* encode *oracle of total length l_e, q_2 queries to the* verify *oracle, and running in time at most t. Let $r=\{r_i\}_{i=1}^n$ be the response sent to challenge $c=\{c_i\}_{i=1}^n$. If there exists at least one r_i not computed as in the* respond *protocol (i.e., as a linear combination of some blocks with indices derived from c_i), then the probability that the reply is accepted is* $\mathsf{Adv}_H^{\mathsf{h\text{-}mac}}(l_e/k_{\mathsf{disp}}, q_2, t)$.

Proof: Assume that $\mathcal{A}$ is able to fabricate an incorrect reply $r=\{r_i\}_{i=1}^n$ to challenge $c=\{c_i\}_{i=1}^n$. We construct an adversary $\mathcal{A}'$ for the homomorphic MAC.

$\mathcal{A}'$ runs $\mathcal{A}$. When $\mathcal{A}$ makes a query to encode of length l, $\mathcal{A}'$ makes l/k_{disp} queries to the homomorphic MAC oracle. When $\mathcal{A}$ makes a query to verify, $\mathcal{A}'$ makes a query to its composite MAC verification oracle. When $\mathcal{A}$ outputs r, $\mathcal{A}'$ outputs r as a composite MAC on $(r_1,\ldots,r_{k_{\mathsf{disp}}})$. It follows that the h-mac-advantage of $\mathcal{A}'$ is the success probability of $\mathcal{A}$. ▮

A discussion for choosing the number of challenges in each interval to bound the fraction of incorrect replies to ϵ_c is included in BJO. The following lemma shows the correspondence between the fraction of red blocks ϵ_b and the fraction of correct replies for a red-blue adversary (step 3 in the outline):

Lemma 4 *Consider a red-blue adversary that corrupts one server that responds correctly to a fraction of $1-\epsilon_c$ of challenges in a time interval. If $0<\epsilon_c\le\frac{d_{\mathsf{in}}}{2n_{\mathsf{in}}}$, and there exists an ϵ_b for which $0<\epsilon_b<\frac{d_{\mathsf{out}}}{4n_{\mathsf{out}}}$, then we can obtain an upper bound on ϵ_b (the fraction of red blocks on the server) from the relation $[1-(1-\epsilon_b)^{k_{\mathsf{in}}}]=\frac{2\epsilon_c n_{\mathsf{in}}}{d_{\mathsf{in}}}$.*

This lemma is part of the proof of the main theorem from [7], and we do not elaborate on its proof. We could apply the result in the lemma to compute ϵ_c as a function of ϵ_b.

The properties of the adversarial outer code construction given in BJO are as follows:

Fact 3 *Let (n_s, k_s, d_s) be the parameters of the code used for encoding stripes in the adversarial outer code construction given in [7]. If $\epsilon_b \leq \frac{d_s}{4n_s}$, then the outer code is an $(2\epsilon_b, \delta, 0)$ adversarial code. (We refer the readers to [7] for an exact definition of δ).*

The composition of the outer and dispersal codes in HAIL can be viewed as an adversarial code. Let us now analyze its security (step 5 in the proof outline).

Lemma 5 *If $\epsilon_b \leq \frac{d_s}{4n_s}$, then the encoding of files in the HAIL protocol is an $(\epsilon_b^H = \frac{2k_{\mathsf{disp}}}{n}\epsilon_b, \delta, 0)$ adversarial ECC (and, consequently, an $(2\epsilon_b^H, \delta, 0)$ adversarial erasure code).*

Proof: Consider an adversary $\mathcal{A}$ for the HAIL encoding that outputs two encoded files c_1 and c_2 (i.e., matrices of dimension $m \times n$) at Hamming distance $\epsilon_b^H mn$ with probability bounded by δ. We construct an adversary $\mathcal{A}'$ for the outer code that outputs c_1' and c_2' constructed from only the first k_{disp} columns of the matrix for c_1 and c_2, respectively (We effectively remove the parity blocks of the dispersal code from the encoding).

The Hamming distance between c_1' and c_2' is at most $\epsilon_b^H mn = 2\epsilon_b m k_{\mathsf{disp}}$. But the size of the file encoded under the outer code is mk_{disp}. Then the AECC-advantage of $\mathcal{A}'$ for $\mathsf{ECC}_{\mathsf{out}}$ is the same as that of $\mathcal{A}$ for the HAIL encoding. ▌

We have now all the elements to complete our security analysis.

Theorem 1 *If $\epsilon_b \leq \frac{d_s}{4n_s}$, let $\epsilon_b^H = \frac{2k_{\mathsf{disp}}}{n}\epsilon_b$, ϵ_c^H be the fraction of incorrect replies for a red-blue adversary that sets at most $(2\epsilon_b^H)$ red blocks (computed as in Lemma 4) and n_c^H be the number of challenges that are needed to ensure an ϵ_c^H-adversary. Then:*
$\mathsf{Adv}^{\mathsf{HAIL}}(\pi, q_1, l_e, q_2, t) \leq T\left[\mathsf{Adv}_H^{\mathsf{h\text{-}mac}}(l_e/k_{\mathsf{disp}}, q_2, t)\right] + \delta.$

Proof: Consider a red-blue adversary $\mathcal{A}$ that participates in experiment $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{HAIL}}(\pi)$ making q_1 queries to encode of total length l_e, q_2 queries to verify, and running in time at most t. The advantage of $\mathcal{A}$ is the probability that the file can not be extracted at the end of the experiment. There are two cases to consider:

1. There exists at least one server for which $\mathcal{A}$ marks as red a fraction of more than $2\epsilon_b^H$ blocks at time $t_c \geq 0$. By Lemmas 3 and 4, the probability that $\mathcal{A}$ still replies correctly to a fraction of $1 - \epsilon_c^H$ challenges in each subsequent time interval is upper bounded by $(T - t_c)\left[\mathsf{Adv}_H^{\mathsf{h\text{-}mac}}(l_e/k_{\mathsf{disp}}, q_2, t)\right]$.

2. $\mathcal{A}$ marks a fraction of at most $2\epsilon_b^H$ red blocks in each server. Drawing on the BJO framework, we can extract from the inner code a fraction of at least $1 - 2\epsilon_b^H$ correct blocks (the blue blocks). From Lemma 5, the probability that the full file can not be extracted from the encoding under the outer and dispersal code is bounded by δ.

The theorem follows immediately. ▌

6 Implementation

We implemented file-encoding functions for the (single-server) POR algorithm and for HAIL. The POR requires pre-computed challenge-response check values to be stored with the file (as there is no cross-server redundancy). The number of check values depends on how often the server will be queried, the desired security level, and the expected lifetime of the file. Reflecting practical parameter choices in [7], we present graphs with a security level of $\delta = 10^{-6}$, an $(233, 251, 18)$-outer code and 1000, 10,000, and 30,000 challenges. These correspond respectively to a one challenge a day for three years, one a day for thirty years, and three a day for thirty years.

By comparison, encoding in HAIL does not involve pre-computed check values, but does involve the dispersal code. We consider different parameters for the dispersal code, varying the number of servers from 3 to 15, and the number of redundant servers from 2 to 8. Due to the use of homomorphic MACs in HAIL, we could use in HAIL an outer erasure code, instead of a full error-correcting code, as used in POR. However, the outer code in HAIL needs to be resistant to the same percentage of errors as that of POR. Intuitively, this is due to an adversarial strategy that targets a subset of same rows of the encoded matrix in all servers. For correcting the same percentage of errors, we set the distance of the outer code in HAIL to be 9, i.e, half that of the POR outer code.

The dispersal and outer codes in our implementation are 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., $GF[2^{256}]$. An $(223, 223 + d, d)$ outer code is obtained by encoding messages of size 223, and truncating the codeword to $223 + d$ symbols. To obtain an $(k_{\mathsf{disp}}, n, d_{\mathsf{disp}})$ outer code, the message is padded with zeros to 223 symbols and the resulting codeword is truncated to n symbols. Thus several optimizations are possible, including a native implementation of the dispersal code and optimizations for small message sizes.

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. Test files were stored on, and output files were written to, a 7200 RPM Hitachi 100 GB Parallel-ATA drive with an 8MB buffer. The average latency time for the hard drive is 4.2ms with an average seek time is 10ms.

The file is encoded in a single pass to minimize the cost of disk accesses. To make this feasible, intermediate state in

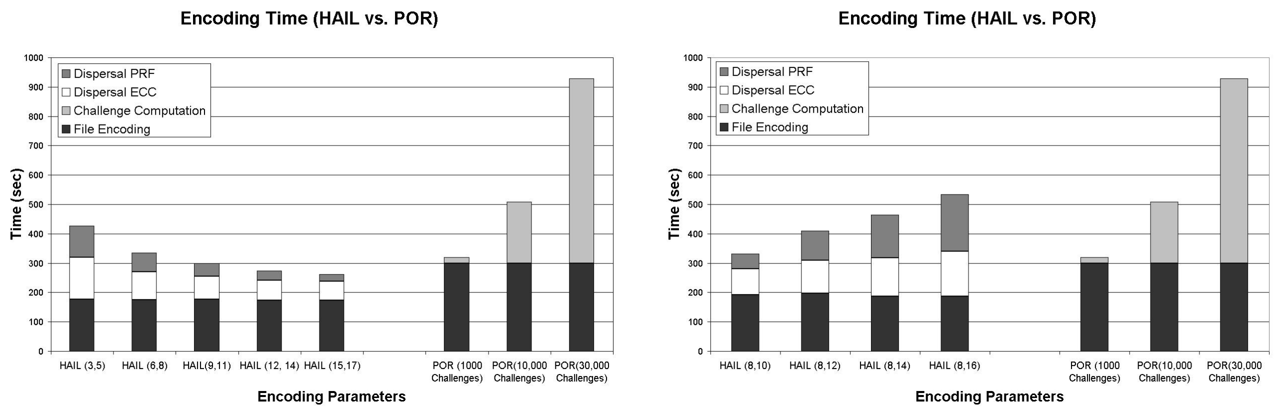

Figure 3. Encoding Time for HAIL vs POR: on the left, constant number of redundant servers; on the right, a constant number of primary servers.

the computation of the outer code must be kept in memory. As shown in [7], this approach encodes 12 GB of file for every 1 GB of available memory. (Larger files may be divided into 12 GB chunks, each requiring separate encoding.)

Figure 3 shows the encoding cost in HAIL compared to POR. On the left graph, we present the encoding cost as the number of redundant servers remains constant at 2, and on the right graph we keep the number of primary k_{disp} servers constant at 8, and increase the redundant servers.

We observe that the outer code encoding in HAIL is reduced by a third compared to POR, due to the smaller distance of the outer code. Total encoding time for HAIL with up to 15 servers is comparable to the cost of POR encoding with 1000 challenges. While the cost of POR dramatically increases with the number of challenges, the cost of HAIL encoding remains the same.

Somewhat counterintuitive is the fact that the dispersal cost decreases as the number of servers increases. This is an artifact of our dispersal code implementation: as k_{disp} increases, there are fewer calls to the Reed-Solomon encoder. We expect that this effect be ameliorated in a code optimized for small message sizes. As expected, when k_{disp} is constant, the dispersal cost increases linearly as more redundant servers are added.

7 Conclusion

We have proposed HAIL, a high-availability and -integrity layer that extends the basic principles of RAID into the more adversarial setting of the cloud. HAIL is a remote-file integrity checking protocol that unifies previous single-server POR protocols with distributed proposals that operate across a set of servers. Through a careful interleaving of different types of error-correcting layers, it protects against a strong, mobile adversary inspired by proactive cryptographic models.

We envision several extensions to HAIL that we plan to address in future work. The first is a model in which servers may be added to or removed from the system across time. An interesting question in such a model is if we can construct efficient protocols for redistribution of shares that do not involve full file reconstruction and redistribution. We also plan to consider adversaries that are not fully Byzantine (erase-only adversaries, economically rational adversaries, etc.), and find efficient HAIL strategies for such adversaries. Finally, we plan to investigate the potential benefits of HAIL protocols that permit inter-server communication.

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] M. Bellare and B. Yee. Forward-security in private-key cryptography. In *CT-RSA '03*, pages –. Springer, 2003. LNCS vol. 2612.

[5] 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.

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

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

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

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

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

[11] 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.

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

[13] 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.

[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] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

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

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

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

[29] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

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

[31] 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.

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

[33] 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.

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

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