New Approaches to Security and Availability for Cloud Data

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

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

1. INTRODUCTION

Cloud computing is a service model that offers users (called herein *tenants*) on-demand network access to a large shared pool of computing resources ("the cloud"). By now the economic benefits of cloud computing are widely recognized. Building and managing a large-scale data center results in savings of a factor between five and seven over a medium-sized one in terms of electricity, hardware, network-bandwidth and operational costs [2]. From the tenant's perspective, the ability to utilize and pay for resources on demand and the rapid elasticity of the cloud are strong incentives for migration to the cloud.

Despite these economic benefits, public clouds still haven't seen widespread adoption, especially by enterprises. Most large organizations today run private clouds, in the sense of virtualized and geographically distributed data centers, but rarely rely primarily on externally managed resources. (Notable exceptions include Twitter and The New York Times, which run on Amazon infrastructure).

Major barriers to adoption are the security and operational risks to which existing cloud infrastructures are prone, including hardware failures, software bugs, power outages, server misconfiguration, malware, and insider threats, among others. Such failures and attack vectors aren't new, but their risk is amplified by the large scale of the cloud [6]. Their impact can be disastrous, and can include data loss and corruption, breaches of data confidentiality, and malicious tampering with data. Therefore, strong protections beyond mere encryption are a necessity for data outsourced to the cloud. Two stand out as particularly important: *integrity*, meaning assurance against data tampering, and *freshness*, the guarantee that retrieved data reflects the latest updates.

Another concern hindering migration into public clouds is a lack of availability and reliability guarantees. Well known cloud providers have experienced episodes of temporary unavailability lasting at least several hours [21, 11] and striking losses of personal customer data (most notably the T-Mobile/Sidekick incident [23]). Traditional reliability models for hardware make certain assumptions about failure patterns (e.g., independence of failures among hard drives) that are not accurate in the new cloud computing world. Without novel data reliability protections (beyond today's RAID-5 and RAID-6, maintaining correctness of massive amounts of data over long periods of time will be extremely difficult [5].

Another top concern for enterprises migrating into the cloud is collocation with potentially malicious tenants [6]. In an Infrastructure-as-a-Service (IaaS) model, tenants rent virtual machines (VMs) on servers shared with other tenants; logical isolation among VMs is enforced by hypervisors. In a Platform-as-a-Service (PaaS) model, different tenants may run applications in the same operating system, without clear isolation beyond basic OS-level protections (that can be easily bypassed by advanced malware available today). Ristenpart et al. [18] show that an attacker can collocate a VM under its control on the same server as a targeted, victim VM in the Amazon IaaS infrastructure. They also give evidence that such an attacker can exploit side channels in shared hardware (e.g., the L2 cache) to exfiltrate sensitive data from the victim.

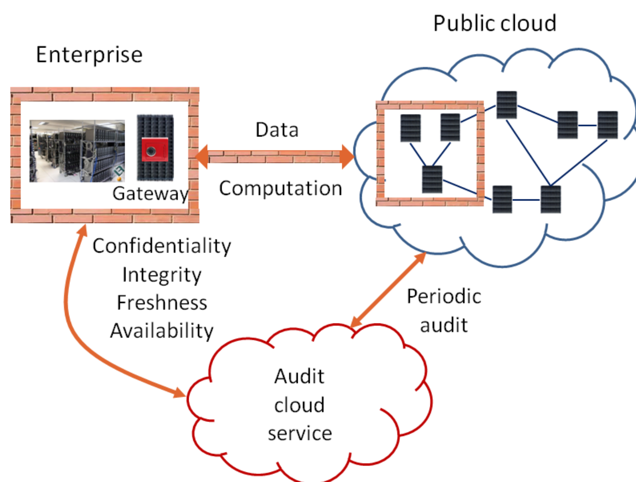

Figure 1: Extending trust perimeter from enterprise data center into the public cloud.

Our research targets the challenge of migrating enterprise data into the public cloud while retaining tenant trust and visibility. We have devised cryptographic protocols that extend traditional trust perimeters from enterprise data centers into the public cloud by conferring strong protections on migrated data, including integrity, freshness, and high availability. In addition, we propose an auditing framework to verify properties of the internal operation of the cloud and assure enterprises that their cloud data—and workloads—are handled securly and reliably.

Adversarial model. In this paper, we are mainly concerned with cloud providers that are subject to a diverse range of threats, but are economically motivated. Cloud providers might deviate from

our protocols for cost savings or due to poor security practices, but not with a pure malicious intent. In most cases, we can detect deviations from our protocols by a misbehaving cloud provider, but we do not provide remediation mechanisms against fully malicious cloud providers. By using multiple cloud providers in the design of HAIL described in Section 4, we show nevertheless that we can also provide data integrity and availability in a setting in which a fraction of providers are fully Byzantine.

Solution overview. Our vision of a more trustworthy cloud computing model for enterprises is depicted graphically in Figure 1. A small trusted *gateway* sitting within the enterprise intermediates all communication from the internal data center to the external public cloud. The gateway manages cryptographic keys (used for encrypting data for confidentiality requirements), maintains trusted storage for integrity and freshness enforcement, and may add redundancy to data for enhanced availability. Once the data and workloads of a particular enterprise migrate into the cloud, an independent cloud auditing service (run by the enterprise or, alternatively, by a third party) continuously monitors the enterprise's cloud resources. This auditing service communicates bidirectionally with the gateway on a regular basis. Updates on enterprise data and workloads migrated into the cloud propagate from the enterprise to the auditing service. The auditing service communicates the results of its audits back to the enterprise, including for instance, scores of the health of various resources (such as data repositories or virtual machines).

Organization. In this paper, we describe several research projects that form components of this broad vision. We start by presenting in Section 2 our design of an *authenticated file system* called Iris that allows migration of existing internal enterprise file systems into the cloud. Iris offers strong integrity and freshness guarantees of both file system data and meta-data accessed while users perform file system operations. Iris is designed to minimize the effects of the network latency on file system operations and carefully optimized for typical file system workloads (sequential file accesses).

We then introduce our auditing framework in Section 3. Within this framework, in Section 3.1 we present *Proofs of Retrievability* (PoR) and related protocols that cryptographically verify the correctness of *all* cloud-stored data with minimal communication. (Remarkably, even against a cheating cloud, PoRs show that *every bit* of the data stored in the cloud is intact.) We describe a dynamic PoR architecture in Section 3.2 that supports data updates in Iris. We turn to audit of physical-layer storage properties in Section 3.3. We show how to verify that cloud data is replicated across multiple hard drives with the RAFT protocol.

For further data protection we address the challenge of data availability in the face of cloud service failures, including potentially malicious ones. In Section 4, we describe HAIL, a protocol that distributes data redundantly across different cloud providers. HAIL is a cloud extension of the RAID principle, building reliable storage systems from inexpensive, unreliable components. We conclude in Section 5 with discussion of some remaining challenges in securing cloud data.

2. INTEGRITY CHECKING WITH IRIS

It's common for tenants to assume that encrypting their data before sending it to the cloud suffices to secure it. Encryption provides strong confidentiality against a prying or breached cloud provider. But it doesn't protect against corruption of data due to software bugs or configuration errors. These challenges require enforcement of a different property:

- *Data integrity* ensures that data retrieved by a tenant is authentic, i.e., hasn't been modified or corrupted by an unauthorized party.

On its own, data integrity is relatively easy to achieve with cryptography (typically by means of Message-Authentication Codes (MACs) on data blocks). But a critical, subtle, related security property of data is often overlooked: *Freshness*. This enures that latest updates are always propagated to the cloud and prevents against rollback attacks (in which stale versions of the data are presented to tenants).

- *Data freshness* ensures that retrieved data always reflects the most recent updates and prevents rollback attacks.

Achieving data freshness is essential to protect against mis-configuration errors or rollbacks caused intentionally. It's the main technical challenge in building the Iris system that we now describe.

2.1 Iris design goals

Iris is an *authenticated file system* that supports the migration of an enterprise-class distributed file system into the cloud efficiently, transparently, and in a scalable manner. It's *authenticated* in the sense that Iris enables an enterprise tenant to verify the integrity and freshness of retrieved data while performing file system operations.

A key design requirement for Iris is that it imposes on client applications no changes to file system operations (file read, write, update, and delete operations, as well as creation and removal of directories). That is, Iris doesn't require user machines to run modified applications. Our design also aims to achieve a slowdown in operation latency small enough to go unnoticed by users even when a large number of clients in the enterprise (on the order of hundreds and even thousands) issue file-system operations in parallel.

2.2 Iris architecture

One of the main challenges we faced when designing Iris is the typically high network latency between an enterprise and the cloud. To reduce the effect of network latency on individual operation latency and the cost of network transfer to and from the cloud, Iris employs heavy caching at the enterprise side.

In Iris, a trusted gateway residing within the enterprise trust boundary (as in Figure 1) intermediates all communication from enterprise users to the cloud. The gateway caches data and meta-data blocks from the file system recently accessed by enterprise users. The gateway computes integrity checks, namely Message Authentication Codes (MAC), on data blocks. It also maintains integrity and freshness information for cached data, consisting of parts of a tree-based authenticated data structure stored on the cloud.

The cloud maintains the distributed file system, consisting of all enterprise files and directories. Iris is designed to use any existing back-end cloud storage system transparently, without modification. The cloud also stores MACs for block-level integrity checks. And it stores a tree-based cryptographic data structure needed to ensure the freshness of data blocks and the directory tree of the file system.

2.3 Integrity and freshness verification

To guarantee data integrity and freshness for the entire file system, Iris employs an authentication scheme consisting of two layers (de-

picted in Figure 2). At the lowest layer, it stores a MAC for each file block (file blocks are fixed-size file segments of typical size 4KB). This enables random access to file blocks and verification of individual file block integrity without accessing full files. For freshness, MACs are not sufficient. Instead, Iris associates a counter or *version number* with each file block that is incremented on every block update (as in [15]) and included in the block MAC. Different versions of a block can be distinguished through different version numbers. But for freshness, block version numbers need to be authenticated too!

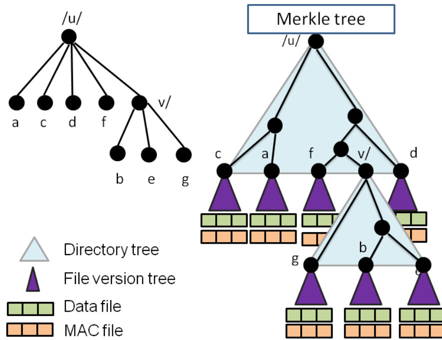

Figure 2: Authentication data structure in Iris.

The upper layer of the authentication scheme is a Merkle tree tailored to the file system directory tree. The leaves of the Merkle tree store block version numbers in a compacted form. The authentication of data is separated from the authentication of block version numbers to enable various optimizations in the data structure. Internal nodes of the tree contain hashes of their children as in a standard Merkle tree. The root of the Merkle tree needs to be maintained at all times within the enterprise trust boundary at the gateway.

The tenant can efficiently verify the integrity and freshness of a file data block by checking the block MAC and the freshness of the block version number. The tenant verifies the latter by accessing the sibling nodes on the path from the leaf storing the version number up to the root of the tree, re-computing all hashes on the path to the root and checking that the root matches the value stored locally. With a similar mechanism, the tenant can additionally verify the correctness of file paths in the file system and, more generally, of any other file system meta-data (file names, number of files in a directory, file creation time, etc.).

This Merkle-tree-based structure has two distinctive features compared to other authenticated file systems: (1) Support for existing file system operations: Iris maintains a balanced binary tree over the file system directory structure to efficiently support existing file system calls; and (2) Support for concurrent operations: The Merkle tree supports efficient updates from multiple clients operating on the file system in parallel. Iris also optimizes for sequential file-block accesses: Sequences of identical version counters are compacted intoa single leaf.

The authentication mechanism in Iris is practical and scalable: in a prototype system, the use of a Merkle tree cache of only 10MB increases the system throughput by a factor of 3 (compared to no caching employed), the throughput is fairly constant for about 100 clients executing operations on the file system in parallel and the operation latency overhead introduced by processing at the gateway (including the integrity checking mechanism) is at most 15%. These numbers are reported from a user-level implementation of Iris evaluated on commonly used benchmarks including IOZone, sequential file reads and writes, and archiving of an entire directory structure [20].

3. AUDITING FRAMEWORK

Tools like Iris enable tenants to deploy their own security protections for data migrated to the cloud. But tenant self-protection is only effective up to a point. Even with Iris in place, for instance, a tenant's data isn't safe against wholesafe service-provider failure. And while Iris enables a tenant to detect data loss resulting from a drive crash, it doesn't give a tenant early warning of the probable precondition: A dangerous lack of provider storage redundancy.

A strong auditing framework is, we believe, the cornerstone of tenant confidence in a service provider. Regulatory, reputational, and contractual assurances of provider safeguards are important. But ongoing technical assurances of solid security are irreplaceable.

Our research envisions a tenant-provider auditing relationship in which a tenant (or an external auditing service acting on tenant's behalf as in Figure 1) can continuously *audit* a provider to prove compliance with a given security policy. The provider responds to a challenge with a compact, real-time proof. Auditing of this kind draws on the structure of a cryptographic challenge-response protocol: The tenant can rigorously verify the provider's response, obtaining a technically strong guarantee of policy compliance.

The challenge-response-style protocols we describe cover a range of security properties, from data correctness to availability in face of provider failures. We give a high-level overview of how they work and what guarantees they offer. We also briefly discuss their place in a larger vision, one in which trusted hardware can complement our auditing framework.

For concreteness, we mimic the cryptographic literature and refer in this section to our canonical tenant as Alice and the cloud provider as Bob. In our description here, Alice also acts as the auditor verifying properties of cloud-stored data, but in our more general framework from Figure 1 the auditing protocol could be executed by a separate entity (the auditing service).

3.1 Auditing data retrievability

When Alice (the tenant) stores data with Bob (the cloud), the most basic assurance she's likely to seek is that her data remains intact. She wants to know that Bob hasn't let her data succumb to bit rot, storage-device failures, corruption by buggy software, or myriad other common threats to data integrity. Because even a well meaning Bob may be vulnerable to infection by malware, Alice also needs such assurance to be robust even if Bob cheats.

One strong, cryptographic approach to such assurance is what's called a Proof of Retrievability (PoR) [12]. A PoR is challenge-response protocol in which Bob proves to Alice that a given piece of data D stored in the cloud is intact and retrievable. While the Iris system enables verification of data integrity for data retrieved from the cloud in the course of performing regular file-system operations, a PoR enables verification of an entire data collection *without retrieving it from the cloud.*

This goal seems at first counterintuitive, even impossible to achieve.

A PoR can demonstrate with a compact proof (on the order of, say, hundreds of bytes) that *every single bit of the data is intact* and accessible to Alice, even if the data is very large (gigabytes or more).

3.1.1 Building a PoR, step by step

To give a sense of how a PoR works, we'll develop a construction in stages. An obvious candidate approach is for Alice simply to store a cryptographic hash $c = h(D)$ of data D in the cloud. To verify that D is intact, then, she challenges Bob to send her c. There are two problems with this idea. First, Bob can cheat. He can store c and throw away D. (A refinement incorporating a secret "nonce" into the hash can address this problem.) Efficiency considerations present a more fundamental drawback: To generate c from D authentically, Bob must hash *all of* D, a resource-intensive process if D is big.

An alternative approach is for Alice to sample data blocks and verify their correctness, in effect spot-checking her data. Let r_i denote the ith data block (blocks are fixed-size segments of data with typical sizes 4KB). Before storing the data into the cloud, Alice locally retains a randomly selected block r_i. To challenge Bob, she sends him the block index i, and asks him to transmit r_i, which she verifies against her local copy. (To amplify the probability of detecting data corruption, Alice can request multiple blocks with independent random indices $i_1, i_2, \ldots, i_m$ simultaneously.)

Now Bob only has to touch a small portion of the data to respond to a challenge, solving the resource-consumption problem with hashing. If D, or a large chunk of D, is missing or corrupted, Alice will detect the fact with high probability, as desired. There are still a couple of drawbacks to this scheme, though.

First, while Alice can detect large corruptions with high probability, she is unlikely to detect small corruptions (say, limited bit rot)—even with multiple challenge blocks. Suppose her data has 1,000,000 blocks, and Alice challenges Bob on ten randomly selected blocks. Then the probability of Alice's detecting a one-bit error is less than 0.001%. Second, Alice must use fresh blocks for every challenge, so that Bob can't predict future challenges from past ones. So if Alice plans to challenge Bob many times, she must store a considerable amount of data locally.

To solve the first problem, we can appeal to an *error-correcting code*. This is a technique for adding redundancy to some piece of data D (called *message*), yielding encoded data D^* (called *codeword*). Often D^* is constructed by appending what are called "parity blocks" to the end of D. If a limited portion of the encoded data D^* is corrupted or erased, it's still possible to apply a decoding function to restore the original data D. The expansion ratio ($|D^*|/|D|$) and amount of tolerable corruption depend on the code parameters. For the sake of example, though, we might consider an error-correcting code that expands data by 10% (i.e., $|D^*|/|D| = 1.1$) and can successfully decode provided that at most 10% of the blocks in D^* are corrupted.

Now, if Alice stores D^* instead of D, she is assured that her data will be lost only if a significant fraction of her data is corrupted or erased. Put another way, for a single bit of D to be irretrievably corrupted, a large chunk of D^* must be. Use of error-correcting effectively amplifies the power of Alice's challenges. Suppose, for instance, that she uses a code that can correct up to 10% corruption. Now, issuing ten challenges against a 1,000,000-block data collection will result in detection of any irretrievably corrupted bits in D with probability more than 65%—a vast improvement over 0.001%!

Solving the problem of excessive local storage is fairly easy. Using a locally stored secret key κ, Alice can compute message authentication codes (MACs) —secret-key digital signatures—over data blocks $r_1, r_2, \ldots, r_n$. She can safely have Bob store these MACs $c_1, c_2, \ldots, c_n$ alongside D^*. To verify the correctness of a block r_i, Alice uses κ and c_i. So all Alice needs to store is the key κ.

Figure 3 shows the data stored by Alice and Bob in this PoR construction.

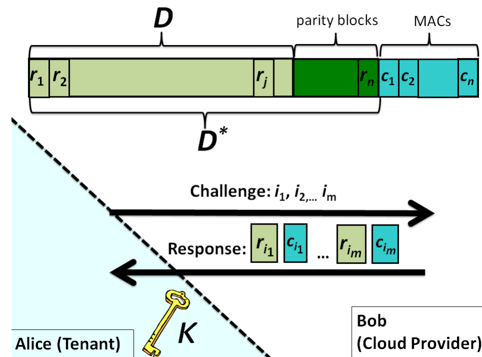

Figure 3: Data stored by Alice and Bob in a PoR. To issue a challenge to Bob, Alice indicates positions $i_1, \ldots, i_m$ to Bob. Bob returns data blocks $r_{i_1}, \ldots, r_{i_m}$, along with MACs $c_{i_1}, \ldots, c_{i_m}$. Alice verifies the correctness of r_{i_j} against MAC c_{i_j}, for all $j \in \{1, \ldots, m\}$ using secret key κ.

There are other challenges in constructing a practical PoR that we haven't discussed. For instance, Alice can send a PRF key during a challenge, based on which Bob can infer the position of all challenged blocks. It's also possible for Bob to aggregate multiple response blocks into one, rendering transmission and verification more efficient. Additionally, making error-correcting practical for large data collections requires some coding and cryptographic tricks. But the solution presented here gives much of the intuition behind a full-blown PoR construction.

A PoR can be used to verify the integrity and correctness of any type of data collection stored in the cloud, including file systems or key-value stores. Its salient property is that it gives Alice strong guarantees about the correctness of an entire data collection using minimal computation and bandwidth for auditing. The auditing protocol could be performed by a third-party auditing service.

3.1.2 Variants

There are several variants for auditing data integrity. A Proof of Data Possession (PDP) [3] enables public verification of data integrity by employing public-key cryptography. Compared with PoRs, PDP provide only detection of large amounts of data corruption, without a recovering mechanism. PDP and PoR protocols can be either privately or publicly verifiable. In a privately-verifiable protocol, the auditing can be performed only by the party that knows the secret key used for encoding the data. In contrast, in a publicly-verifiable protocol, auditing can be performed by any third-party service, at the cost of more computationally expensive encoding and auditing protocols. The term *Proof of Storage* (PoS) [4] has evolved as a convenient catchall term for PoRs and PDPs. Most

PoS protocols can be combined with other cryptographic protections, e.g., encryption for data confidentiality, to achieve a suite of cloud data protections [13].

3.2 PoRs in Iris

A basic PoR, as described above, has one notable limitation: It doesn't handle data updates gracefully. Changing a single data block in D requires propagation of changes across the parity blocks of D^*. So a basic PoR is only efficient for checks on *static* data, such as archived data. The situation is somewhat better without error correction; researchers have proposed (asymptotically efficient) PDP systems that support data updates [9]. But then support for updates rules out recovering from small data corruption.

It's natural, then, to ask whether we can have the best of both worlds: A PoR for dynamically changing data. The answer is that we can.

Check values (MACs or digital signatures) pose the first major challenge in supporting a dynamic PoR / PDP. Not only must they be updated in conjunction with data-block updates, but when Alice verifies them, she must be able to determine both that they're correct and that they're fresh.

The Iris system, as explained above, is designed precisely to tackle the tricky problem of efficient freshness verification for file systems. So it's an ideally suited platform for building a dynamic PoR.

An even more formidable challenge, though, is that of updating error-correcting blocks as data blocks change. Briefly, to protect against targeted corruption by Bob, the structure of the error-correcting code, and thus the the pattern of parity block updates, must be hidden from him. Encryption doesn't help: Basic encryption hides data, not access patterns.

The way out of this conundrum is a model shift, inspired by the deployment objectives of Iris. While PoR designers generally aim to keep Alice's storage to a minimum, Iris aims at enterprise-class cloud deployments. When Alice is a company, not an individual, substantial tenant-side resources are a reasonable expectation.

Thus the key idea for dynamic PoR layered on Iris: Have Alice cache parity blocks locally, on the enterprise side, and periodically back them up to the cloud. This approach conceals individual parity block updates from Bob and hides the code structure. It has another advantage too. Alice's updates to parity blocks can be made locally. As a single data block update results in multiple parity-block updates, the ability to make updates locally greatly reduces communication between Alice to Bob.

The end result is an enhancement such that Iris not only verifies file integrity and freshness, but can also check efficiently whether an entire file system is intact—down to the last bit. In addition, if corruptions to data are found (either through auditing or through integrity verification using the Merkle tree structure described in Section 2), Iris can recover corrupted blocks from the additional redundancy provided by the erasure code. Thus, Iris provides strong guarantees of detection and remediation of data corruption, resulting in retrievability of an entire file system stored in the cloud. A great benefit of Iris is that its parameters for the erasure code and the communication during an audit can be adjusted for a desired level of recoverability.

While the construction in Iris provides the first practical solution to a dynamic PoR protocol, it relies on some amount of local storage maintained by the tenant (in our Iris instantiation the client maintains $O(\sqrt{n})$ local storage). The problem of constructing a dynamic PoR protocol with constant storage at the client side is the major remaining theoretical research challenge.

3.3 Auditing of drive-failure resilience

Detecting data loss via Proofs-of-Storage is helpful. Preventing it would be better. One of the major causes of data loss is drive crashes. In a large data center with hundreds of thousands of drives, drive failures is the norm rather than the exception. With 2-3% annual failure rates published by manufactures and even higher numbers observed in the field [19], a large data center experiences thousands of drive failures every year.

Services such as Amazon S3 claim to store files in triplicate. How can this claim be verified remotely? At first glance, it seems impossible. Suppose Alice wants to verify that Bob is storing three copies of her data. Downloading the three copies obviously won't work: If Bob is cheating and storing just one copy of the data, he can simply transmit that one copy three times!

There's a very simple solution. Alice can encrypt each copy of her data under a separate key, yielding three distinct encryptions. Executing a PoS against each encrypted version, then, ensures the existence of three distinct copies.

In many cases, though, Alice doesn't want to have to store her data in encrypted form. She may want Bob to be able to process her data for her or make it available to her friends. More importantly, the existence of three distinct copies *per se* doesn't ensure resilience to drive crashes. All three copies could be sitting on the same drive, after all. So the real problem is this: *How can Alice verify that there are three distinct copies of the data, each on a different drive?*

A striking feature of this problem is that it's primarily physical, not logical. The objective is not to verify the encoding or mere existence of data, but its disposition on a physical substrate.

A system called RAFT (Remote Assessment of Fault Tolerance) [8] solves this problem. It allows Alice to verify that Bob has stored some piece of data D so that it can survive up to t drive failures, for a desired parameter t. RAFT allows D to be dispersed using erasure coding, a more space-efficient technique than maintaining full file copies.

RAFT operates specifically on data stored in rotational drives. It exploits the performance limitations of these drives as a bounding parameter. The more drives across which Bob has striped D, the faster he can respond to a challenge. In particular, RAFT makes use of bounds on the *seek time* of a rotational drive. Alice transforms her data D using an erasure code into encoded data D^*. D^* can be striped across c drives such that if any t fail, it's possible to recover D. She asks Bob to store D^* across c drives.

To verify resilience to t drive failures, Alice challenges Bob to fetch a set of n randomly distributed blocks from D^*. Suppose that Bob stores D^* on d drives. Each block fetch incurs a seek (assuming that the random blocks are spread apart at large distance). So on average, if a seek takes time μ, Bob's total fetch time is $\mu n/d$. If $d < c$, then his response time will take $\mu n(1/d - 1/c)$ longer than expected, on average. By measuring Bob's response time, then,

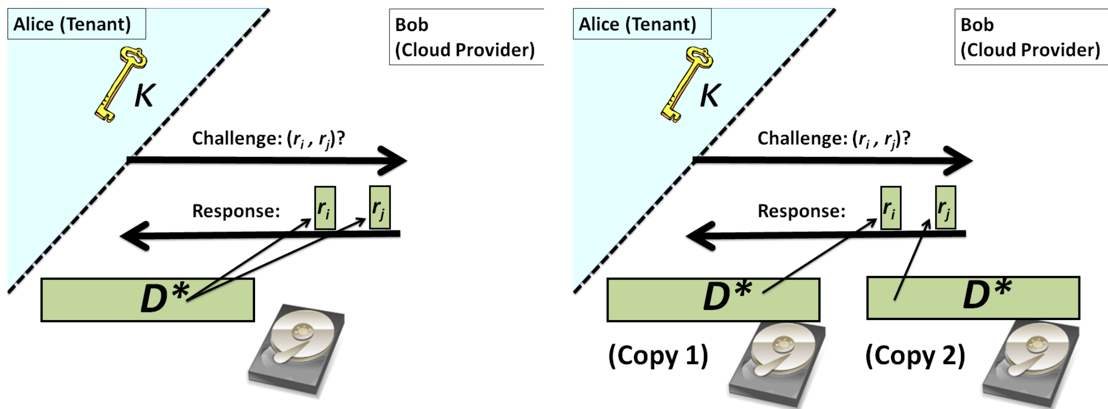

Figure 4: Responding to challenges from one disk on the left and two disks on the right.

Alice can determine whether he's indeed using c drives, as required. A graphical representation is given in Figure 4.

While we do not go into many details here, in a real-world setting, many complications arise. Variations in drive performance and also in network latency between Alice and Bob need to be carefully measured. Sensitive statistical analysis and structuring of challenges is required to accommodate these variations.

3.4 Hardware roots of trust

Another approach to assurance within the challenge-response framework explored here is use of a *hardware root of trust*, as supported, for instance, by Trusted Platform Modules (TPMs). These permit a tenant to verify remotely, via a challenge-response protocol, that a provider is executing a particular software stack.

But hardware roots of trust *can't* directly enforce guarantees on storage integrity or reliability. Even if Bob's servers are configured precisely as specified by Alice, and even if Alice controls their operation, she obtains no direct guarantees about their corresponding storage subsystem. For instance, the only way for Alice to determine that a file F is intact in storage without full-file inspection remains to perform a PoR. The same holds for properties verified by Iris, HAIL, and RAFT: they can not be guaranteed solely by use of trustworthy execution environments.

4. ENHANCING DATA AVAILABILITY WITH HAIL

We've described an auditing framework that offers tenants visibility into the operations of the cloud and verification of some properties of their cloud-side data. But what happens if the cloud provider fails to respond correctly to an audit because of data loss? A major impediment to cloud adoption by enterprises is the danger of provider temporary unavailability or even permanent failure. This is a real threat, as illustrated by catastrophic provider failures resulting in customer data loss [23].

We have designed a system called HAIL [7] (High-Availability and Integrity Layer) specifically to address this challenge. HAIL is predicated on the idea that it's wise to distribute data across multiple cloud providers for continuous availability. HAIL thus leverages multiple cloud providers to construct a reliable and cost-effective cloud storage service out of unreliable components. This idea is similar in flavor to RAID [16]. RAID creates reliable storage arrays from unreliable hard drives. HAIL extends this idea into the cloud, the main differences being its support of a stronger adversarial model and a higher-level abstraction.

HAIL works by promptly detecting and recovering from data corruption. The tenant (or a third-party service) periodically audits individual cloud providers toward this end. HAIL auditing is lightweight in terms of both bandwidth and computation. Using the redundancy embedded across different cloud providers, the tenant (or third-party) remediates corruptions detected in a subset of providers. HAIL is reactive, rather than proactive, meaning that it only remediates data upon detected corruption.

4.1 System model

In HAIL a tenant distributes her data with embedded redundancy to a set of n cloud providers: $S_1, \ldots, S_n$. In our model, data generated by all enterprise users is transmitted to the gateway (as in Figure 1). The gateway performs some data *encoding*, described below, optionally encrypts data, and distributes a data fragment to each cloud provider.

HAIL operates in an adversarial model in which a strong *mobile* adversary can corrupt *all* cloud providers over time. But within a single epoch (a pre-determined period of fixed length) the adversary can corrupt at most b out of n servers, for some $b < n$.

4.2 HAIL encoding

The encoding of data in HAIL is depicted in Figure 5. To provide resilience against cloud provider failure, the gateway splits the data into fixed-sized blocks, and encodes the data with a new erasure code called *dispersal code*. Figure 5 shows a matrix-representation of the data on the left, which results in an encoded matrix on the right. Each row in the encoded matrix is a *stripe* or codeword obtained by applying the dispersal code. It contains the original data blocks, as well as new parity blocks obtained with the dispersal code. Each matrix column is stored at a different cloud provider. The dispersal code guarantees that the original data can be reconstructed given up to b cloud provider failures (and $n - b$ intact columns).

A single layer of encoding, however, doesn't suffice in HAIL's strong adversarial model. Consider the following attack: the adversary corrupts b new servers in each epoch, picks a particular row index i and corrupts the corresponding block $r_{i,j}$ at server S_j. After $\lceil n/b \rceil$ epochs, the adversary corrupts all servers and the en-

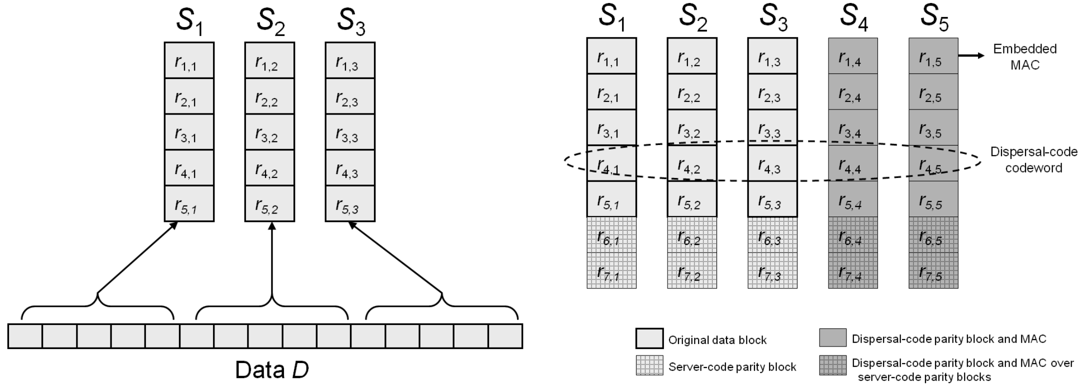

Figure 5: Encoding of data D: on the left, original data represented as a matrix; on the right, encoded data with parity blocks added for both the server and dispersal codes.

tire row i in the encoded matrix from Figure 5. In this case, the redundancy of the dispersal code is not helpful in recovering the corrupted row and the entire data D is permanently corrupted.

How can this creeping-corruption attack be prevented? By simply auditing a few randomly selected blocks at each server, the probability that the tenant discovers the corruption of blocks in a single row of the encoded matrix is very low. For this reason, *within* each server is needed another encoding layer, called a *server code*. The server code adds additional redundancy (parity blocks) to each column in the encoded matrix representation. The role of the server code is to recover from a small amount of corruption at each cloud provider, undetectable through the auditing protocol.

To prevent adversarial data corruption, the tenant also needs to store MACs on data blocks in the cloud. With a new technique called an *integrity-protected dispersal code*, it's possible to use parity blocks of the dispersal code themselves as MACs on the rows, and thus reduce the storage overhead for integrity protection.

4.3 Auditing and recovering from failures

In HAIL, the gateway (or an external auditing service as shown in Figure 1) periodically audits the correctness of the cloud data, by contacting all cloud providers. The gateway sends a random row index i as a challenge to each cloud provider, and verifies, upon receiving the responses $r_{i,j}$, for $j \in \{1, \ldots, n\}$ the correctness of the entire row. It's also possible to aggregate multiple responses (multiple randomly selected blocks) from each server to reduce bandwidth and amplify the probability of failure detection.

When data corruption at one or several cloud providers is detected, the corrupted data can be reconstructed at the tenant side using the two encoding layers: the dispersal and server code. Data reconstruction is an expensive protocol, one that will be rarely invoked (only upon detection of data corruption).

With its encoding, auditing and recovery mechanisms, HAIL provides resilience against a strong mobile adversary that can potentially corrupt all providers over time. A limitation of HAIL is that as designed, it doesn't gracefully handle file updates. It's most suited to archival data, data stored in the cloud for retention purposes and not regularly modified. We believe that more efficient versions of HAIL can be constructed under a weaker adversarial model that may be practical for short-term data storage.

5. CONCLUSIONS AND OPEN PROBLEMS

We have described new techniques that secure cloud data by ensuring a range of protections from integrity and freshness verification to high data availability. We've also proposed an auditing framework that offers tenants visibility into the correct operation of the cloud. These techniques enable an extension of the trust perimeter from enterprise internal data centers into public clouds as shown in Figure 1. It is our hope that these techniques will alleviate some of the concerns around security in the cloud and facilitate the migration of enterprise resources into public clouds.

We conclude by mentioning of some remaining issues and open problems of interest in this context:

Performing computations over tenants' encrypted data Our emphasis here has been on data integrity and availability, but data confidentiality remains a major open problem. General computation over a tenant's encrypted data is possible using a technique called fully-homomorphic encryption (FHE). A recent breakthrough [10], FHE is not yet practical. Weaker, custom techniques can achieve specific functionalities, though, such as searches [14] and general SQL queries [17] over encrypted data.

The impossibility of general computations over multiple tenants' encrypted data using only cryptographic techniques and no interaction among tenants is shown in [22]. A promising area of research is the design of custom protocols for applications involving multiple tenants' data, e.g., data mining over multiple institutions' medical records or financial transactions. We believe that combining secure hardware architectures with cryptography (e.g., secure multi-party computation protocols) offers huge potential.

Ensuring tenant isolation Cloud co-tenancy with attackers can jeopardize tenant data, as shown in [18] which explores cache-based side channels for data exfiltration. The risks of co-tenancy in storage systems, e.g., storage-system side channels, is an unexplored vector of attack deserving investigation in our view.

One approach to co-tenancy risks is to isolate tenants by implementing Virtual Private Cloud (VPC) abstractions within a public cloud. HomeAlone [24] enables a tenant to verify remotely that a VPC is strongly enforced at the host level, in the sense of creating physical isolation of a tenant's workloads. While this offers a solution for extremely sensitive workloads, it can undercut the

economic benefits of tenant sharing of computing resources. For this reason, we believe that solutions offering tenant isolation and enabling sharing of cloud resources at the same time are extremely important. Trusted hardware may play an important role, as well as tight enforcement of logical isolation abstractions throughout the software stack (hypervisor, OS, etc.) and across the cloud fabric.

Geolocation of data An open problem of particular commercial interest is remote verification of the geographical location of cloud data. The motivation is regulatory compliance: many laws require providers to keep customer data within national boundaries [1], for instance. Given the challenge of ensuring that data isn't duplicated, any solution will probably require a trusted data management plane (via, e.g., trusted hardware). Localizing the pieces of this plane, though, is an interesting challenge. Geolocation of trusted hardware via remote timing from trusted anchor points seems a key avenue of exploration.

6. REFERENCES

[1] Directive 95/46/EC of the European Parliament of the Council ("Data Protection Directive"), 1995. `http://bit.ly/5eLDdi`.

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[4] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology - ASIACRYPT '09*, 2009.

[5] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long term digital storage. In *Proc. European Conference on Computer Systems (EuroSys)*, 1996.

[6] M. Blumenthal. Is security lost in the cloud? *Communications and Strategies, The Economics of Cybersecurity*, (81), 2011.

[7] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[8] K. D. Bowers, M. van Dijk, A. Juels, A. Oprea, and R. L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[9] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[10] C. Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3), 2010.

[11] M. Helft. Google confirms problems with reaching its services, 2009. `http://www.developmentguruji.com/news/99/Google-confirms-problems-with-reaching-its-services.html`.

[12] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[13] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Workshop on Real-Life Cryptographic Protocols and Standardization*, 2010.

[14] S. Kamara, C. Papamanthou, and T. Roeder. Cs2: A searchable cryptographic cloud storage system. Technical Report MSR-TR-2011-58, Microsoft, 2011.

[15] A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proc. 16th Usenix Security Symposium*, 2007.

[16] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM International Conference on Management of Data (SIGMOD)*, 1988.

[17] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[18] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proc. 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[19] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proc. 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[20] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks, 2011. IACR ePrint manuscript 2011/585.

[21] A. Stern. Update from Amazon regarding Friday S3 downtime, 2008. `http://www.centernetworks.com/amazon-s3-downtime-update`.

[22] M. van Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proc. HOTSEC*, 2010.

[23] N. Wingfield. Microsoft, T-Mobile stumble with Sidekick glitch. The Wall Street Journal, 2011. `http://online.wsj.com/article/SB10001424052748703790404574467431941990194.html`.

[24] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proc. IEEE Symposium on Security and Privacy*, 2011.