How to Tell if Your Cloud Files Are Vulnerable to Drive Crashes

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

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

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

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

Ronald L. Rivest
MIT CSAIL
Cambridge, MA, USA
rivest@mit.edu

ABSTRACT

This paper presents a new challenge—verifying that a remote server is storing a file in a fault-tolerant manner, i.e., such that it can survive hard-drive failures. We describe an approach called the *Remote Assessment of Fault Tolerance* (RAFT). The key technique in a RAFT is to measure the *time taken* for a server to respond to a read request for a collection of file blocks. The larger the number of hard drives across which a file is distributed, the faster the read-request response. Erasure codes also play an important role in our solution. We describe a theoretical framework for RAFTs and offer experimental evidence that RAFTs can work in practice in several settings of interest.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Cloud storage, auditing, fault tolerance, erasure codes

1. INTRODUCTION

Cloud storage offers clients a unified view of a file as a single, integral object. This abstraction is appealingly simple. In reality, though, cloud providers generally store files/objects with redundancy or error correction to protect against data loss. Amazon, for example, claims that its S3 service stores three replicas of each object[1]. Additionally, cloud providers often spread files across multiple storage devices. Such distribution provides resilience against hardware failures, e.g., drive crashes (and can also lower latency across disparate geographies).

The single-copy file abstraction in cloud storage, however, conceals file-layout information from clients. It therefore deprives them of insight into the true degree of fault-tolerance their files enjoy. Even when cloud providers specify a storage policy (e.g., even given Amazon's claim of triplicate storage), clients have no technical means of *verifying* that their files aren't vulnerable, for instance, to drive crashes. In light of clients' increasingly critical reliance on cloud storage for file retention, and the massive cloud-storage failures that have come to light, e.g., [9], it is our belief that remote testing of fault tolerance is a vital complement to contractual assurances and service-level specifications.

In this paper we develop a protocol for remote testing of fault-tolerance for stored files. We call our approach the *Remote Assessment of Fault Tolerance* (RAFT). A RAFT enables a client to obtain proof that a given file F is distributed across physical storage devices to achieve a certain desired level of fault tolerance. We refer to storage units as *drives* for concreteness. For protocol parameter t, our techniques enable a cloud provider to prove to a client that the file F can be reconstructed from surviving data given failure of any set of t drives. For example, if Amazon were to prove that it stores a file F fully in triplicate, i.e., one copy on three distinct drives, this would imply that F is resilient to $t = 2$ drive crashes.

At first glance, proving that file data is stored redundantly, and thus proving fault-tolerance, might seem an impossible task. It is straightforward for storage service $\mathcal{S}$ to prove knowledge of a file F, and hence that it has stored at least one copy. $\mathcal{S}$ can just transmit F. But how can $\mathcal{S}$ prove, for instance, that it has *three distinct copies* of F? Transmitting F three times clearly doesn't do the trick! Even proving storage of three copies doesn't prove fault-tolerance: the three copies could all be stored on the same disk!

To show that F isn't vulnerable to drive crashes, it is necessary to show that it is spread across multiple drives. Our approach, the Remote Assessment of Fault Tolerance, proves the use of multiple drives by exploiting drives' performance constraints—in particular bounds on the *time* required for drives to perform challenge tasks. A RAFT is structured as a timed challenge-response protocol. A short story gives the intuition. Here, the aim is to ensure that a pizza order can tolerate $t = 1$ oven failures.

> A fraternity ("Eeta Pizza Pi") regularly orders pizza from a local pizza shop, "Cheapskate Pizza." Recently Cheapskate failed to deliver pizzas for the big pregame party, claiming that their only pizza oven had suffered

[1]Amazon has also recently introduced reduced redundancy storage that promises less fault tolerance at lower cost

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-0948-6/11/10 ...$10.00.

a catastrophic failure. They are currently replacing it with two new BakesALot ovens, for increased capacity and reliability in case one should fail.

Aim O'Bese, president of Eeta Pizza Pi, wants to verify that Cheapskate has indeed installed redundant pizza ovens, without having to visit the Cheapskate premises himself. He devises the following clever approach. Knowing that each BakesALot oven can bake two pizzas every ten minutes, he places an order for two dozen pizzas, for delivery to the fraternity as soon as possible. Such a large order should take an hour of oven time in the two ovens, while a single oven would take two hours. The order includes various unusual combinations of ingredients, such as pineapple, anchovies, and garlic, to prevent Cheapskate from delivering warmed up pre-made pizzas.

Cheapskate is a fifteen minute drive from the fraternity. When Cheapskate delivers the two dozen pizzas in an hour and twenty minutes, Aim decides, while consuming the last slice of pineapple/anchovy/garlic pizza, that Cheapskate must be telling the truth. He gives them the fraternity's next pregame party order.

Our RAFT for drive fault-tolerance testing follows the approach illustrated in this story. The client challenges the server to retrieve a set of random file blocks from file F. By responding quickly enough, $\mathcal{S}$ proves that it has distributed F across a certain, minimum number of drives. Suppose, for example, that $\mathcal{S}$ is challenged to pull 100 random blocks from F, and that this task takes one second on a single drive. If $\mathcal{S}$ can respond in only half a second[2], it is clear that it has distributed F across at least two drives.

Again, the goal of a RAFT is for $\mathcal{S}$ to prove to a client that F is recoverable in the face of at least t drive failures for some t. Thus $\mathcal{S}$ must actually do more than prove that F is distributed across a certain number of drives. It must *also* prove that F has been stored with a certain amount of *redundancy* and that the distribution of F across drives is *well balanced*. To ensure these two additional properties, the client and server agree upon a particular mapping of file blocks to drives. An underlying erasure code provides redundancy. By randomly challenging the server to show that blocks of F are laid out on drives in the agreed-upon mapping, the client can then verify resilience to t drive failures.

The real-world behavior of hard drives presents a protocol-design challenge: The response time of a drive can vary considerably from read to read. Our protocols rely in particular on timing measurements of disk seeks, the operation of locating randomly accessed blocks on a drive. Seek times exhibit high variance, with multiple factors at play (including disk prefetching algorithms, disk internal buffer sizes, physical layout of accessed blocks, etc.). To smooth out this variance we craft our RAFTs to sample *multiple* randomly selected file blocks per drive. Clients not only check the correctness of the server's responses, but also measure response times and accept a proof only if the server replies within a certain time interval.

We propose and experimentally validate on a local system a RAFT that can, for example, distinguish between a three-drive system with fault tolerance $t = 1$ and a two-drive system with no fault tolerance for files of size at least 100MB. Additionally, we explore the feasibility of the RAFT protocol on the Mozy cloud backup system and confirm that Mozy is resilient to at least one drive failure. We conclude that RAFT protocols are most applicable to test fault tolerance for large files in an archival or backup setting in which files are infrequently accessed and thus there is limited drive contention.

Our RAFT protocol presented in this paper is designed for traditional storage architectures that employ disk-level replication of files and use hard disk drives (HDDs) as the storage medium. While these architectures are still prevalent today, there are many settings in which our current protocol design is not directly applicable. For instance, the characteristics of HDD's sequential and random access do not hold for SSD drives or RAM memory, which could potentially be used for performance-sensitive workloads in systems employing multi-tier storage (e.g., [2, 28]). Systems with data layout done at the block level (as opposed to file-level layout) are also not amenable to our current design, as in that setting timing information in our challenge-response protocol does not directly translate into fault tolerance. Examples of architectures with block-level data layout are chunk-based file systems [1, 15], and systems with block-level de-duplication [8]. Other features such as spinning disks down for power savings, or replicating data across different geographical locations complicate the design of a RAFT-like protocol. Nevertheless, we believe that our techniques can be adapted to some of these emerging architectures and we plan to evaluate this further in future work.

RAFTs aim primarily to protect against "economically rational" service providers/adversaries, which we define formally below. Our adversarial model is thus a mild one. We envision scenarios in which a service provider agrees to furnish a certain degree of fault tolerance, but cuts corners. To reduce operating costs, the provider might maintain equipment poorly, resulting in unremediated data loss, enforce less file redundancy than promised, or use fewer drives than needed to achieve the promised level of fault tolerance. (The provider might even use too few drives accidentally, as virtualization of devices causes unintended consolidation of physical drives.) An economically rational service provider, though, *only* provides substandard fault tolerance when doing so reduces costs. The provider does not otherwise, i.e., maliciously, introduce drive-failure vulnerabilities. We explain later, in fact, why protection against malicious providers is technically infeasible.

1.1 Related work

Proofs of Data Possession (PDPs) [3] and Proofs of Retrievability (PORs) [10, 11, 21, 31] are challenge-response protocols that verify the integrity and completeness of a remotely stored F. They share with our work the idea of combining error-coding with random sampling to achieve a low-bandwidth proof of storage of a file F. This technique was first advanced in a theoretical framework in [27]. Both [23] and [5] remotely verify fault tolerance at a logical level by using multiple independent cloud storage providers. A RAFT includes the extra dimension of verifying physical layout of F and tolerance to a number of drive failures at a single provider.

Cryptographic challenge-response protocols prove knowledge of a secret—or, in the case of PDPs and PORs, knowledge of a file. The idea of timing a response to measure remote physical resources arises in cryptographic *puzzle* constructions [12]. For instance, a challenge-response protocol based on a moderately hard computational problems can measure the computational resources of clients submitting service requests and mitigate denial-of-service attacks by proportionately scaling service fulfilment [20]. Our protocols here measure not computational resources, but the storage

[2] Of course, $\mathcal{S}$ can violate our assumed bounds on drive performance by employing unexpectedly high-grade storage devices, e.g., flash storage instead of rotational disks. As we explain below, though, our techniques aim to protect against economically rational adversaries $\mathcal{S}$. Such an $\mathcal{S}$ might create substandard fault tolerance to cut costs, but would not employ more expensive hardware just to circumvent our protocols. (More expensive drives often mean higher reliability anyway.)

resources devoted to a file. (Less directly related is physical distance bounding, introduced in a cryptographic setting in [7]. There, packet time-of-flight gives an upper bound on distance.)

We focus on non-Byzantine, i.e., non-malicious, adversarial models. We presume that malicious behavior in cloud storage providers is rare. As we explain, such behavior is largely irremediable anyway. Instead, we focus on an adversarial model ("cheap-and-lazy") that captures the behavior of a basic, cost-cutting or sloppy storage provider. We also consider an economically rational model for the provider. Most study of economically rational players in cryptography is in the multiplayer setting, but economical rationality is also implicit in some protocols for storage integrity. For example, [3,16] verify that a provider has dedicated a certain amount of storage to a file F, but don't strongly assure file integrity. We formalize the concept of self-interested storage providers in our work here.

A RAFT falls under the broad heading of cloud security assurance. There have been many proposals to verify the security characteristics and configuration of cloud systems by means of trusted hardware, e.g., [14]. Our RAFT approach advantageously avoids the complexity of trusted hardware. Drives typically don't carry trusted hardware in any case, and higher layers of a storage subsystem can't provide the physical-layer assurances we aim at here.

1.2 Organization

Section 2 gives an overview of key ideas and techniques in our RAFT scheme. We present formal adversarial and system models in Section 3. In Section 4, we introduce a basic RAFT in a simple system model. Drawing on experiments, we refine this system model in Section 5, resulting in a more sophisticated RAFT which we validate against the Mozy cloud backup service in Section 6. In Section 7, we formalize an economically rational adversarial model and sketch matching RAFT constructions. We conclude in Section 8 with discussion of future directions.

2. OVERVIEW: BUILDING A RAFT

We now discuss in further detail the practical technical challenges in building a RAFT for hard drives, and the techniques we use to address them. We view the file F as a sequence of m blocks of fixed size (e.g., 64KB).

File redundancy / erasure coding. To tolerate drive failures, the file F must be stored with redundancy. A RAFT thus includes an initial step that expands F into an n-block erasure-coded representation G. If the goal is to place file blocks evenly across c drives to tolerate the failure of any t drives, then we need $n = mc/(c - t)$. Our adversarial model, though, also allows the server to drop a portion of blocks or place some blocks on the wrong drives. We show how to parameterize our erasure coding at a still higher rate, i.e., choose a larger n, to handle these possibilities.

Challenge structure. (*"What order should Eeta Pizza Pie place to challenge Cheapskate?"*) We focus on a "layout specified" RAFT, one in which the client and server agree upon an exact placement of the blocks of G on c drives, i.e., a mapping of each block to a given drive. The client, then, challenges the server with a query Q that selects exactly one block per drive in the agreed-upon layout. An honest server can respond by pulling exactly one block per drive (in one "step"). A cheating server, e.g., one that uses fewer than c drives, will need at least one drive to service *two* block requests to fulfill Q, resulting in a slowdown.

Network latency. (*"What if Cheapskate's delivery truck runs into traffic congestion?"*) The network latency, i.e., roundtrip packet-travel time, between the client and server, can vary due to changing network congestion conditions. The client cannot tell how much a response delay is due to network conditions and how much might be due to cheating by the server. Based on the study by Lumezanu et al [24] and our own small-scale experiments, we set an upper bound threshold on the variability in latency between a challenging client and a storage service. We consider that time to be "free time" for the adversary, time in which the adversary can cheat, prefetching blocks from disk or perform any other action that increases his success probability in the challenge-response protocol. We design our protocol to be resilient to a bounded amount of "free time" given to the adversary.

Drive read-time variance. (*"What if the BakesALot ovens bake at inconsistent rates?"*) The read-response time for a drive varies across reads. We perform experiments, though, showing that for a carefully calibrated file-block size, the response time follows a probability distribution that is stable across time and physical file positioning on disk. (We show how to exploit the fact that a drive's "seek time" distribution is stable, even though its read bandwidth isn't.) We also show how to smooth out read-time variance by constructing RAFT queries Q that consist of *multiple* blocks per drive.

Queries with multiple blocks per drive. (*"How can Eeta Pizza Pie place multiple, unpredictable orders without phoning Cheapskate multiple times?"*) A naïve way to construct a challenge Q consisting of multiple blocks per drive (say, q) is simply for the client to specify cq random blocks in Q. The problem with this approach is that the server can then schedule the set of cq block accesses on its drives to reduce total access time (e.g., exploiting drive efficiencies on sequential-order reads). Alternatively, the client could issue challenges in q steps, waiting to receive a response before issuing a next, unpredictable challenge. But this would require c rounds of interaction.

We instead introduce an approach that we call *lock-step* challenge generation. The key idea is for the client to specify query Q in an initial step consisting of c random challenge blocks (one per drive). For each subsequent step, the set of c challenge blocks depends on the *content* of the file blocks accessed in the last step. The server can proceed to the next step only after fully completing the last one. Our lock-step technique is a kind of repeated application of a Fiat-Shamir-like heuristic [13] for generating q independent, unpredictable sets of challenges non-interactively. (File blocks serve as a kind of "commitment.") The server's response to Q is the aggregate (hash) of all of the cq file blocks it accesses.

3. FORMAL DEFINITIONS

A Remote Assessment of Fault Tolerance $\mathcal{RAFT}(t)$ aims to enable a service provider to prove to a client that it has stored file F with tolerance against t drive failures. In our model, the file is first encoded by adding some redundancy. This can be done by either the client or the server. The encoded file is then stored by the server using some number of drives. Periodically, the client issues challenges to the server, consisting of a subset of file block indices. If the server replies correctly and promptly to challenges (i.e., the answer is consistent with the original file F, and the timing of the response is within an acceptable interval), the client is convinced that the server stores the file with tolerance against t drive failures. The client can also reconstruct the file at any time from the encoding stored by the server, assuming at most t drive failures.

3.1 System definition

To define our system more formally, we start by introducing some notation. A file block is an element in $B = GF[2^\ell]$. For

convenience we also treat ℓ as a security parameter. We let f_i denote the ith block of a file F for $i \in \{1, \ldots, |F|\}$.

The RAFT system comprises these functions:

- $\mathsf{Keygen}(1^\ell) \xrightarrow{R} \kappa$: A key-generation function that outputs key κ. We denote a keyless system by $\kappa = \phi$.

- $\mathsf{Encode}(\kappa, F = \{f_i\}_{i=1}^m, t, c) \rightarrow G = \{g_i\}_{i=1}^n$: An encoding function applied to an m-block file $F = \{f_i\}_{i=1}^m$; it takes as additional input fault tolerance t and a number of c logical disks. It outputs encoded file $G = \{g_i\}_{i=1}^n$, where $n \geq m$. The function Encode may be keyed, e.g., encrypting blocks under κ, in which case encoding is done by the client, or unkeyed, e.g., applying an erasure code or keyless cryptographic operation to F, which may be done by either the client or server.

- $\mathsf{Map}(n, t, c) \rightarrow \{C_j\}_{j=1}^c$: A function computed by both the client and server that takes the encoded file size n, fault tolerance t and a number c of logical disks and outputs a logical mapping of file blocks to c disks or $\perp$. To implement Map the client and server can agree on a mapping which each can compute, or the server may specify a mapping that the client verifies as being tolerant to t drive failures. (A more general definition might also include $G = \{g_i\}_{i=1}^n$ as input. Here we only consider mappings that respect erasure-coding structure.) The output consists of sets $C_j \subseteq \{1, 2, \ldots, n\}$ denoting the block indices stored on drive j, for $j \in \{1, \ldots, c\}$. If the output is not $\perp$, then the placement is tolerant to t drive failures.

- $\mathsf{Challenge}(n, G, t, c) \rightarrow Q$: A (stateful and probabilistic) function computed by the client that takes as input the encoded file size n, encoded file G, fault tolerance t, and the number of logical drives c and generates a challenge Q consisting of a set of block indices in G and a random nonce ν. The aim of the challenge is to verify disk-failure tolerance at least t.

- $\mathsf{Response}(Q) \rightarrow (R, T)$: An algorithm that computes a server's response R to challenge Q, using the encoded file blocks stored on the server disks. The timing of the response T is measured by the client as the time required to receive the response from the server after sending a challenge.

- $\mathsf{Verify}(G, Q, R, T) \rightarrow b \in \{0, 1\}$: A verification function for a server's response (R, T) to a challenge Q, where 1 denotes "accept," i.e., the client has successfully verified correct storage by the server. Conversely 0 denotes "reject." Input G is optional in some systems.

- $\mathsf{Reconstruct}(\kappa, r, \{g_i^*\}_{i=1}^r) \rightarrow F^* = \{f_i^*\}_{i=1}^m$: A reconstruction function that takes a set of r encoded file blocks and either reconstructs an m-block file or outputs failure symbol $\perp$. We assume that the block indices in the encoded file are also given to the Reconstruct algorithm, but we omit them here for simplicity of notation. The function is keyed if Encode is keyed, and unkeyed otherwise.

Except in the case of Keygen, which is always probabilistic, functions may be probabilistic or deterministic. We define $\mathcal{RAFT}(t) = \{\mathsf{Keygen}, \mathsf{Encode}, \mathsf{Map}, \mathsf{Challenge}, \mathsf{Response}, \mathsf{Verify}, \mathsf{Reconstruct}\}$.

3.2 Client model

In some instances of our protocols called *keyed protocols*, the client needs to store secret keys used for encoding and reconstructing the file. *Unkeyed protocols* do not make use of secret keys for file encoding, but instead use public transforms.

If the Map function outputs a logical layout $\{C_j\}_{j=1}^c \neq \perp$, then we call the model *layout-specified*. We denote a *layout-free* model one in which the Map function outputs $\perp$, i.e., the client does not know a logical placement of the file on c disks. In this paper, we only consider layout-specified protocols, although layout-free protocols are an interesting point in the RAFT design space.

For simplicity in designing the Verify protocol, we assume that the client keeps a copy of F locally. Our protocols can be extended easily via standard block-authentication techniques, e.g., [25], to a model in which the file is maintained only by the provider and the client deletes the local copy after outsourcing the file.

3.3 Drive and network models

The response time T of the server to a challenge Q as measured by the client has two components: (1) Drive read-request delays and (2) Network latency. We model these two protocol-timing components as follows.

Modeling drives.

We model a server's storage resources for F as a collection of d independent hard drives. Each drive stores a collection of file blocks. The drives are stateful: The timing of a read-request response depends on the query history for the drive, reflecting block-retrieval delays. For example, a drive's response time is lower for sequentially indexed queries than for randomly indexed ones, which induce seek-time delays [30]. We do not consider other forms of storage here, e.g., solid-state drives[3].

We assume that all the drives belong to the same class, (e.g. enterprise class drives), but may differ in significant ways, including seek time, latency, and even manufacturer. We will present disk access time distributions for several enterprise class drive models. We also assume that when retrieving disk blocks for responding to client queries in the protocol, there is no other workload running concurrently on the drive, i.e. the drive has been "reserved" for the RAFT[4]. Alternatively, in many cases, drive contention can be overcome by issuing more queries. We will demonstrate the feasibility of both approaches through experimental results.

Modeling network latency.

We adapt our protocols to handle variations in network latency between the client and cloud provider. Based on the results presented in [24] and our own small-scale experiments, we set an upper bound threshold on the variability in observed latency between the vast majority of host pairs. We design our protocol so that the difference in timing to reply successfully to a challenge between an

[3]At the time of this writing, SSDs are still considerably more expensive than rotational drives and have much lower capacities. A typical rotational drive can be bought for roughly \$0.07/GB in capacities up to 3 TBs, while most SSDs cost more than \$2.00/GB are are only a few hundred GBs in size. For an economically rational adversary, the current cost difference makes SSDs impractical.

[4]Multiple concurrent workloads could skew disk-access times in unexpected ways. This was actually seen in our own experiments when the OS contended with our tests for access to a disk, causing spikes in recorded read times.In a multi-tenant environment, users are accustomed to delayed responses, so reserving a drive for 500 ms. to perform a RAFT test should not be an issue.

adversarial and an honest server is at least the maximum observed variability in network latency.

3.4 Adversarial model

We now describe our adversarial model, i.e., the range of behaviors of $\mathcal{S}$. In our model, the m-block file F is chosen uniformly at random. This reflects our assumption that file blocks are already compressed by the client, for storage and transmission efficiency, and also because our RAFT constructions benefit from random-looking file blocks. Encode is applied to F, yielding an encoded file G of size n, which is stored with $\mathcal{S}$.

Both the client and server compute the logical placement $\{C_j\}_{j=1}^c$ by applying the Map function. The server then distributes the blocks of G across d real disks. The number of actual disks d used by $\mathcal{S}$ might be different than the number of agreed-upon drives c. The actual file placement $\{\mathcal{D}_j\}_{j=1}^d$ performed by the server might also deviate arbitrarily from the placement specified by the Map function. (As we discuss later, sophisticated adversaries might even store something other than unmodified blocks of G.)

At the beginning of a protocol execution, we assume that no blocks of G are stored in the high-speed (non-disk) memory of $\mathcal{S}$. Therefore, to respond to a challenge Q, $\mathcal{S}$ must query its disks to retrieve file blocks. The variable T denotes the time required for the client, after transmitting its challenge, to receive a response R from $\mathcal{S}$. Time T includes both network latency and drive access time (as well as any delay introduced by $\mathcal{S}$ cheating).

The goal of the client is to establish whether the file placement implemented by the server is resilient to at least t drive failures.

Our adversarial model is validated by the design and implementation of the Mozy online backup system, which we will discuss further in Section 6.2. We expect Mozy to be representative of many cloud storage infrastructures.

Cheap-and-lazy server model.

For simplicity and realism, we focus first on a restricted adversary $\mathcal{S}$ that we call *cheap-and-lazy*. The objective of a cheap-and-lazy adversary is to reduce its resource costs; in that sense it is "cheap." It is "lazy" in the sense that it does not modify file contents. The adversary instead cuts corners by storing less redundant data on a smaller number of disks or mapping file blocks unevenly across disks, i.e., it may ignore the output of Map. A cheap-and-lazy adversary captures the behavior of a typical cost-cutting or negligent storage service provider.

To be precise, we specify a cheap-and-lazy server $\mathcal{S}$ by the following assumptions on the blocks of file F:

- **Block obliviousness:** The behavior of $\mathcal{S}$ i.e., its choice of internal file-block placement $(d, \{\mathcal{D}_j\}_{j=1}^d)$ is independent of the content of blocks in G. Intuitively, this means that $\mathcal{S}$ doesn't inspect block contents when placing encoded file blocks on drives.

- **Block atomicity:** The server handles file blocks as atomic data elements, i.e., it doesn't partition blocks across multiple storage devices.

A cheap-and-lazy server may be viewed as selecting a mapping from n encoded file blocks to positions on d drives without knowledge of G. Some of the encoded file blocks might not be stored to drives at all (corresponding to dropping of file blocks), and some might be duplicated onto multiple drives. $\mathcal{S}$ then applies this mapping to the n blocks of G.

General adversarial model.

It is also useful to consider a general adversarial model, cast in an experimental framework. We define the security of our system $\mathcal{RAFT}(t)$ according to the experiment from Figure 1. We let $\mathcal{O}(\kappa) = \{\mathsf{Encode}(\kappa,\cdot,\cdot,\cdot), \mathsf{Map}(\cdot,\cdot,\cdot), \mathsf{Challenge}(\cdot,\cdot,\cdot,\cdot), \mathsf{Verify}(\cdot,\cdot,\cdot,\cdot), \mathsf{Reconstruct}(\kappa,\cdot,\cdot)\}$ denote a set of RAFT-function oracles (some keyed) accessible to $\mathcal{S}$.

Experiment $\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m,\ell,t)$:
  $\kappa \leftarrow \mathsf{Keygen}(1^\ell)$;
  $F = \{f_i\}_{i=1}^m \leftarrow_R B^m$;
  $G = \{g_i\}_{i=1}^n \leftarrow \mathsf{Encode}(\kappa, F, t, c)$;
  $(d, \{\mathcal{D}_j\}_{j=1}^d) \leftarrow \mathcal{S}^{\mathcal{O}(\kappa)}(n, G, t, c,$ "store file");
  $Q \leftarrow \mathsf{Challenge}(n, G, t, c)$;
  $(R, T) \leftarrow \mathcal{S}^{\{\mathcal{D}_j\}_{j=1}^d}(Q,$ "compute response");
  if $\mathsf{Acc}_{\mathcal{S}}$ and $\mathsf{NotFT}_{\mathcal{S}}$
    then output 1,
  else output 0

Figure 1: Security experiment

We denote by $\mathsf{Acc}_{\mathcal{S}}$ the event that $\mathsf{Verify}(G,Q,R,T) = 1$ in a given run of $\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m,\ell,t)$, i.e., that the client / verifier accepts the response of $\mathcal{S}$. We denote by $\mathsf{NotFT}_{\mathcal{S}}$ the event that there exists $\{\mathcal{D}_{i_j}\}_{j=1}^{d-t} \subseteq \{\mathcal{D}_j\}_{j=1}^d$ s.t. $\mathsf{Reconstruct}(\kappa, |\{\mathcal{D}_{i_j}\}_{j=1}^{d-t}|, \{\mathcal{D}_{i_j}\}_{j=1}^{d-t}) \neq F$, i.e, the allocation of blocks selected by $\mathcal{S}$ in the experimental run is not t-fault tolerant.

We define $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m,\ell,t) = \Pr[\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m,\ell,t) = 1] = \Pr[\mathsf{Acc}_{\mathcal{S}} \text{ and } \mathsf{NotFT}_{\mathcal{S}}]$. We define the *completeness* of $\mathcal{RAFT}(t)$ as $\mathsf{Comp}^{\mathcal{RAFT}(t)}(m,\ell,t) = \Pr[\mathsf{Acc}_{\mathcal{S}} \text{ and } \neg\mathsf{NotFT}_{\mathcal{S}}]$ over executions of honest $\mathcal{S}$ (a server that always respects the protocol specification) in $\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m,\ell,t)$.

Our general definition here is, in fact, a little too general for practical purposes. As we now explain, there is no good RAFT for a fully malicious $\mathcal{S}$. That is why we restrict our attention to cheap-and-lazy $\mathcal{S}$, and later, in Section 7, briefly consider a "rational" $\mathcal{S}$.

Why we exclude malicious servers.

A *malicious* or fully Byzantine server $\mathcal{S}$ is one that may expend arbitrarily large resources and manipulate and store G in an arbitrary manner. Its goal is to achieve $\leq t-1$ fault tolerance for F while convincing the client with high probability that F enjoys full t fault tolerance.

We do not consider malicious servers because there is no efficient protocol to detect them. A malicious server can convert any t-fault-tolerant file placement into a 0-fault-tolerant file placement very simply. The server randomly selects an encryption key λ, and encrypts every stored file block under λ. $\mathcal{S}$ then adds a new drive and stores λ on it. To reply to a challenge, $\mathcal{S}$ retrieves λ and decrypts any file blocks in its response. If the drive containing λ fails, of course, the file F will be lost. There is no efficient protocol that distinguishes between a file stored encrypted with the key held on a single drive, and a file stored as specified, as they result in nearly equivalent block read times.[5]

3.5 Problem Instances

A RAFT *problem instance* comprises a client model, an adversarial model, and drive and network models. In what follows, we

[5]The need to pull λ from the additional drive may slightly skew the response time of $\mathcal{S}$ when first challenged by the client. This skew is modest in realistic settings. And once read, λ is available for any additional challenges.

propose RAFT designs in an incremental manner, starting with a very simple problem instance—a cheap-and-lazy adversarial model and simplistic drive and network models. After experimentally exploring more realistic network and drive models, we propose a more complex RAFT. We then consider a more powerful ("rational") adversary and further refinements to our RAFT scheme.

4. THE BASIC RAFT PROTOCOL

In this section, we construct a simple RAFT system resilient against the cheap-and-lazy adversary. We consider very simple disk and network models. While the protocol presented in this section is mostly of theoretical interest, it offers a conceptual framework for later, more sophisticated RAFTs.

We consider the following problem instance:

Client model: Unkeyed and layout-specified.

Adversarial model: The server is cheap-and-lazy.

Drive model: Time to read a block of fixed length ℓ from disk is constant and denoted by τ_ℓ.

Network model: The latency between client and server (denoted L) is constant in time and network bandwidth is unlimited.

4.1 Scheme Description

To review: Our RAFT construction encodes the entire m-block file F with an erasure code that tolerates a certain fraction of block losses. The server then spreads the encoded file blocks evenly over c drives and specifies a layout. To determine that the server respects this layout, the client requests c blocks of the file in a challenge, one from each drive. The server should be able to access the blocks in parallel from c drives, and respond to a query in time close to $\tau_\ell + \mathsf{L}$.

If the server answers most queries correctly and promptly, then blocks are spread out on disks almost evenly. A rigorous formalization of this idea leads to a bound on the fraction of file blocks that are stored on any t server drives. If the parameters of the erasure code are chosen to tolerate that amount of data loss, then the scheme is resilient against t drive failures.

To give a formal definition of the construction, we use a maximum distance separable (MDS), i.e., optimal erasure code with encoding and decoding algorithms $(\mathsf{ECEnc}, \mathsf{ECDec})$ and expansion rate $1 + \alpha$. ECEnc encodes m-block messages into n-block codewords, with $n = m(1 + \alpha)$. ECDec can recover the original message given any αm erasures in the codeword.

The scheme is the following:

- $\mathsf{Keygen}(1^\ell)$ outputs ϕ.
- $\mathsf{Encode}(\kappa, F = \{f_i\}_{i=1}^m, t, c)$ outputs $G = \{g_i\}_{i=1}^n$ with n a multiple of c and $G = \mathsf{ECEnc}(F)$.
- $\mathsf{Map}(n, t, c)$ outputs a balanced placement $\{C_j\}_{j=1}^c$, with $|C_j| = n/c$. In addition $\cup_{j=1}^c C_j = \{1, \ldots, n\}$, so consequently $C_i \cap C_j = \phi, \forall i \neq j$.
- $\mathsf{Challenge}(n, G, t, c)$ outputs $Q = \{i_1, \ldots, i_c\}$ consisting of c block indices, each i_j chosen uniformly at random from C_j, for $j \in \{1, \ldots, c\}$. (Here, we omit nonce ν.)
- $\mathsf{Response}(Q)$ outputs the response R consisting of the c file blocks specified by Q, and the timing T measured by the client.
- $\mathsf{Verify}(G, Q, R, T)$ performs two checks. First, it checks *correctness* of blocks returned in R using the file stored locally by the client[6]. Second, the client also checks the *promptness* of the reply. If the server replies within an interval $\tau_\ell + \mathsf{L}$, the client outputs 1.
- $\mathsf{Reconstruct}(\kappa, r, \{g_i^*\}_{i=1}^r)$ outputs the decoding of the file blocks retained by $\mathcal{S}$ (after a possible drive failure) under the erasure code: $\mathsf{ECDec}(\{g_i^*\}_{i=1}^r)$ for $r \geq m$, and $\perp$ if $r < m$.

The security analysis of the protocol is deferred to the full version of the paper [6]. Here we summarize the main result.

THEOREM 1. *For fixed system parameters c, t and α such that $\alpha \geq t/(c - t)$ and for constant network latency and constant block read time, the protocol satisfies the following properties for a cheap-and-lazy server $\mathcal{S}$:*
1. The protocol is complete: $\mathsf{Comp}^{\mathcal{RAFT}(t)}(m, \ell, t) = 1$.
2. If $\mathcal{S}$ uses $d < c$ drives, $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t) = 0$.
3. If $\mathcal{S}$ uses $d \geq c$ drives, $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t) \leq 1 - B(c, t, \alpha)$ *where* $B(c, t, \alpha) = \frac{\alpha(c-t)-t}{(1+\alpha)(c-t)}$.

Multiple-step protocols.

We can make use of standard probability amplification techniques to further reduce the advantage of a server. For example, we can run multiple steps of the protocol. A step for the client involves sending a c-block challenge, and receiving and verifying the server response. We need to ensure that queried blocks are different in all steps, so that the server cannot reuse the result of a previous step in successfully answering a query.

We define two queries Q and Q' to be *non-overlapping* if $Q \cap Q' = \emptyset$. To ensure that queries are non-overlapping, the client running an instance of a multiple-step protocol maintains state and issues only queries with block indices not used in previous query steps. We can easily extend the proof of Theorem 1 (3) to show that a q-step protocol with non-overlapping queries satisfies $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t) \leq (1 - B(c, t, \alpha))^q$ for a server $\mathcal{S}$ using $d \geq c$ drives.

5. NETWORK AND DRIVE TIMING MODEL

In the simple model of Section 4, we assume constant network latency between the client and server and a constant block-read time. Consequently, for a given query Q, the response time of the server (whether honest or adversarial) is deterministic. In practice, though, network latencies and block read times are variable. In this section, we present experiments and protocol-design techniques that can be used to adapt our simple RAFT protocol to more practical settings.

5.1 Network model

We present some experimental data on network latency between hosts in different geographical locations based on the Lumezanu et al. study [24], and quantify the amount of variance it exhibits over time. We discuss how our RAFT protocol can be made robust against variability in network latency. We also show how to reduce the communication complexity of our protocol—thereby eliminating network-timing variance due to fluctuations in network bandwidth.

[6] Recall we assume a copy of the file is kept by the client to simplify verification, though this is not necessary.

Network latency model.

Lumezanu et al. [24] present a study that measures the network latency between 1715 pairs of hosts at various time intervals within a two month period. Their goal is to study the occurrence and characteristics of triangle inequality violations in the Internet. In one of their experiments, they measure the variability of network latency among pairs of hosts over time. Their findings indicate that for about 88% of host pairs in their trace, the standard deviation of network latency is less than 100ms. Another metric of variability is the inter-quartile range of the latency distribution (defined as the difference between 75th and 25th percentiles). They show that less than 10% of the host pairs have inter-quartile higher than 40ms, suggesting that the variance in network latency is caused by outliers farther away from the mean, rather than values closer to the mean.

For our purposes, we are interested in estimating the maximum difference between round-trip times observed at various times between the same pair of hosts. While the study does not give us directly an estimate for this metric, we can approximate the 99th percentile of the difference, for instance, by three standard deviations, i.e., 300ms (which for the normal distribution cover 99.7% of the distribution).

To validate this choice of parameters, we perform our own small-scale experiments. We pinged two hosts (one in Santa Clara, CA, USA and one in Shanghai, China) from our Boston, MA, USA location during a one week interval in March 2010. We observed that the ping time distribution is heavy tailed with spikes correlated in time, most likely due to temporary network congestion.

The ping times to Santa Clara ranged from 86 ms to 463 ms, with 90th, 99th and 99.9th percentiles at 88ms, 95ms and 102ms, respectively. Ping times to Shanghai exhibit more variability across the larger geographical distance and range between 262 ms and 724 ms. While daily spikes in latency raise the average slightly, 90% of readings are still less than 278 ms. These spikes materially lengthen the tail of the distribution, however, as the 99% (433 ms) and 99.9% (530 ms) thresholds are no longer grouped near the 90% mark, but are instead much more spread out. To summarize, the 99th percentile of the difference in network latency is 9 ms for Santa Clara and 171 ms for Shanghai. The 99.9th percentile results in 16ms for Santa Clara, and 268ms for Shanghai. We believe therefore that a choice of three standard deviations (300ms) is a reasonable maximum variability in network latency we can set in our RAFT experiments.

In our RAFT protocol, we consider a response valid if it arrives within the maximum characterized network latency. We then adopt the bounding assumption that the difference between minimum and maximum network latency is "free time" for an adversarial server. That is, during a period of low latency, the adversary might simulate high latency, using the delay to cheat by prefetching file blocks from disk into cache. This strategy would help the server respond to subsequent protocol queries faster, and help conceal poor file-block placement. If the amount of data which can be read during this "free time" is small compared to the size of the file, the effect is insignificant. We quantify this precisely in the full version of the paper [6].

Limited network bandwidth.

In the basic protocol from Section 4, challenged blocks are returned to the client as part of the server's response. To minimize the bandwidth used in the protocol, the server can simply apply a cryptographically strong hash to its response blocks together with a nonce supplied by the client, and return the resulting digest. The client can still verify the response, by recomputing the hash value locally and comparing it with the response received from the server.

5.2 Drive model

We now look to build a model for the timing characteristics of magnetic hard drives. While block read times exhibit high variability due to both physical factors and prefetching mechanisms, we show that for a judicious choice of block size (64KB on a typical drive), read times adhere to a stable probability distribution. This observation yields a practical drive model for RAFT.

Drive characteristics.

Magnetic hard drives are complex mechanical devices consisting of multiple platters rotating on a central spindle at speeds of up to 15,000 RPM for high-end drives today. The data is written and read from each platter with the help of a disk head sensing magnetic flux variation on the platter's surface. Each platter stores data in a series of concentric circles, called tracks, divided further into a set of fixed-size (512 byte) sectors. Outer tracks store more sectors than inner tracks, and have higher associated data transfer rates.

To read or write to a particular disk sector, the drive must first perform a *seek*, meaning that it positions the head on the right track and sector within the track. Disk manufacturers report average seek times on the order of 2 ms to 15 ms in today's drives. Actual seek times, however, are highly dependent on patterns of disk head movement. For instance, to read file blocks laid out in sequence on disk, only one seek is required: That for the sector associated with the first block; subsequent reads involve minimal head movement. In constrast, random block accesses incur a highly variable seek time, a fact we exploit for our RAFT construction.

After the head is positioned over the desired sector, the data is read from the platter. The data transfer rate (or throughput) depends on several factors, but is on the order of 300MB per second for high-end drives. The disk controller maintains an internal cache and implements complex caching and prefetching policies. As drive manufacturers give no clear specifications of these policies, it is difficult to build general data access models for drives [30].

The numbers we present in this paper are derived from experiments performed on a number of enterprise class SAS drives, all connected to a single machine running Red Hat Enterprise Linux WS v5.3 x86_64. We experimented with drives from Fujitsu, Hitachi, HP[7], and Seagate. Complete specifications for each drive can be found in Table 1.

Modeling disk-access time.

Our basic RAFT protocol is designed for blocks of fixed-size, and assumes that block read time is constant. In reality, though block read times are highly variable, and depend on both physical file layout and drive-read history. Two complications are particularly salient: (1) Throughput is highly dependent on the absolute physical position of file blocks on disk; in fact, outer tracks exhibit up to 30% higher transfer rates than inner tracks [29] and (2) The transfer rate for a series of file blocks depends upon their relative position; reading of sequentially positioned file blocks requires no seek, and is hence much faster than for scattered blocks.

We are able, however, to eliminate both of these sources of read-time variation from our RAFT protocol. The key idea is to *render seek time the dominant factor* in a block access time. We accomplish this in two ways: (1) We read *small* blocks, so that seek time

[7]Upon further inspection, the HP drive is actually manufactured by Seagate. Nearly all drives available today are in fact made by one of three manufacturers.

Manufacturer	Model	Capacity	Buffer Size	Avg. Seek / Full Stroke Seek	Latency	Throughput
Hitachi	HUS153014VLS300	147 GB	16 MB	3.4 ms./ 6.5 ms.	2.0 ms.	72 - 123 MB/sec
Seagate	ST3146356SS	146 GB	16 MB	3.4 ms./6.43 ms.	2.0 ms.	112 - 171 MB/sec
Fujitsu	MBA3073RC	73.5 GB	16 MB	3.4 ms./8.0 ms.	2.0 ms.	188 MB/sec
HP	ST3300657SS	300 GB	16 MB	3.4 ms./6.6 ms.	2.0 ms.	122 - 204 MB/sec

Table 1: Drive specifications

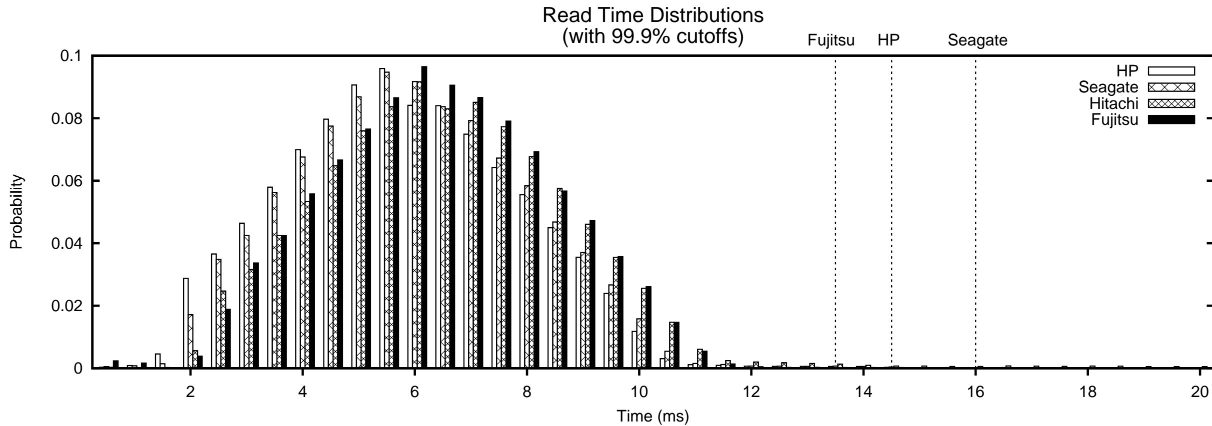

Figure 2: Read time distribution for 64KB blocks

dominates read time and (2) We access a *random* pattern of file blocks, to force the drive to perform a seek of comparable difficulty for each block.

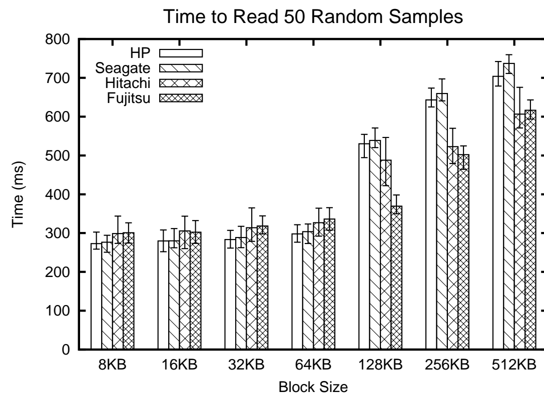

Figure 3: Read time for 50 random blocks

As Figure 3 shows, the time to sample a fixed number of random blocks from a 2GB file is roughly constant for blocks up to 64KB, regardless of drive manufacturer. We suspect that this behavior is due to prefetching at both the OS and hard drive level. Riedel et al. also observe in their study [29] that the OS issues requests to disks for blocks of logical size 64KB, and there is no noticeable difference in the time to read blocks up to 64KB.

For our purposes, therefore, a remote server can read 64KB random blocks at about the same speed as 8K blocks. If we were to sample blocks smaller than 64KB in our RAFT protocol, we would give an advantage to the server, in that it could prefetch some additional file blocks essentially for free. For this reason, we choose to use 64KB blocks in our practical protocol instantiation.

Figure 2 depicts the read time distributions for a random 64KB block chosen from a 2GB file. To generate this distribution, 250 random samples were taken from a 2GB file. The read time for each request was recorded. This was repeated 400 times, for a total of 100,000 samples, clearing the system memory and drive buffer between each test. The operating system resides on the Hitachi drive, and occasionally contends for drive access. This causes outliers in the tests (runs which exceed 125% of average and contain several sequential reads an order of magnitude larger than average), which were removed. Additional tests were performed on this drive to ensure the correctness of the results. By comparison, the variability between runs on all other drives was less than 10%, further supporting the OS-contention theory.

While the seek time average for a single block is around 6 ms, the distribution exhibits a long tail, with values as large as 132 ms. (We truncate the graph at 20 ms for legibility.) This long tail does not make up a large fraction of the data, as indicated by the 99.9% cutoffs in figure 2, for most of the drives. The 99.9% cutoff for the Hitachi drive is not pictured as it doesn't occur until 38 ms. Again, we expect contention from the OS to be to blame for this larger fraction of slow reads on that drive.

Read times for blocks of this size are dominated by seek time and not affected by physical placement on disk. We confirmed this experimentally by sampling from many files at different locations on disk. Average read times between files at different locations differed by less than 10%. The average seek time for 64KB blocks does, however, depend on the size of the file from which samples are being taken, as shown in Figure 4.

We observe that the average block read time increases with the file size, due to more head movement. While this relationship is fairly linear above a certain point (close to 40MB files), small files exhibit significantly reduced average block read times, likely due to the disk buffer. Once the file is small enough to fit in the disk buffer, the drive will respond from its cache without performing the

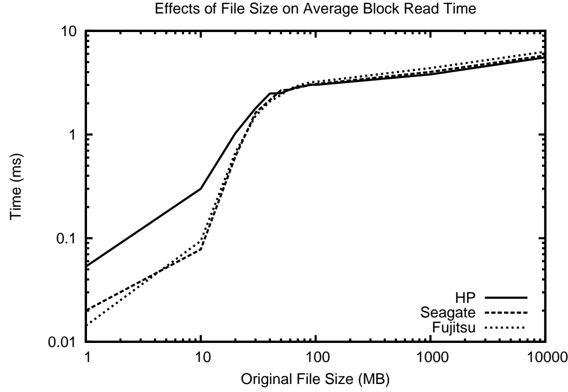

Figure 4: Effects of file size on average block retrieval time

physical seek, returning data much more quickly. This indicates that RAFTs will not work for files smaller than the combined disk buffer sizes of the drives being used to store the file, an hypothesis we confirm in our experimental evaluation.

In the next section, we modify our basic protocol to smooth out seek-time variance. The idea is to sample (seek) many randomly chosen file blocks in succession.

6. PRACTICAL RAFT PROTOCOL

In this section, we propose a practical variant of the basic RAFT protocol from Section 4. As discussed, the main challenge in practical settings is the high variability in drive seek time. The key idea in our practical RAFT here is to smooth out the block access-time variability by requiring the server to access multiple blocks per drive to respond to a challenge.

In particular, we structure queries here in multiple *steps*, where a step consists of a set of file blocks arranged such that an (honest) server must fetch one block from each drive. We propose in this section what we call a *lock-step protocol* for disk-block scheduling. This lock-step protocol is a non-interactive, multiple-step variant of the basic RAFT protocol from Section 4. We show experimentally that for large enough files, the client can, with high probability, distinguish between a correct server and an adversarial one.

6.1 The lock-step protocol

A naïve approach to implementing a multiple-step protocol with q steps would be for the client to generate q (non-overlapping) challenges, each consisting of c block indices, and send all qc distinct block indices to the server. The problem with this approach is that it immediately reveals complete information to the server about all queries. By analogy with job-shop scheduling [26], the server can then map blocks to drives to shave down its response time. In particular, it can take advantage of drive efficiencies on reads ordered by increasing logical block address [32]. Our lock-step technique reveals query structure incrementally, and thus avoids giving the server an advantage in read scheduling. Another possible approach to creating a multi-step query would be for the client to specify steps interactively, i.e., specify the blocks in step $i+1$ when the server has responded to step i. That would create high round complexity, though. The benefit of our lock-step approach is that it generates steps unpredictably, but non-interactively.

The lock-step approach works as follows. The client sends an initial one-step challenge consisting of c blocks, as in the basic RAFT protocol. As mentioned above, to generate subsequent steps non-interactively, we use a Fiat-Shamir-like heuristic [13] for signature schemes: The block indices challenged in the next step depend on all the block contents retrieved in the current step (a "commitment"). To ensure that block indices retrieved in next step are unpredictable to the server, we compute them by applying a cryptographically strong hash function to all block contents retrieved in the current step. The server only sends back to the client the final result of the protocol (computed as a cryptographic hash of all challenged blocks) once the q steps of the protocol are completed.

The lock-step protocol has Keygen, Encode, Map, and Reconstruct algorithms similar to our basic RAFT. Let h be a cryptographically secure hash function with fixed output (e.g., from the SHA-2 family). Assume for simplicity that the logical placement generated by Map in the basic RAFT protocol is $C_j = \{jn/c, jn/c+1, \ldots, jn/c+n/c-1\}$. We use c suitable hash functions that output indices in C_j: $h_j \in \{0,1\}^* \to C_j$. (In practice, we might take $h_j(x) = h(\tilde{j}||x) \mod C_j$, where $\tilde{j}$ is a fixed-length index encoding.)

The Challenge, Response, and Verify algorithms of the lock-step protocol with q steps are the following:

- In Challenge(n, G, t, c), the client sends an initial challenge $Q = (i_1^1, \ldots, i_c^1)$ with each i_j^1 selected randomly from C_j, for $j \in \{1, \ldots, c\}$, along with random nonce $\nu \in_U \{0,1\}^l$.

- Algorithm Response(Q) consists of the following steps:

1. $\mathcal{S}$ reads file blocks $f_{i_1^1}, \ldots, f_{i_c^1}$ specified in Q.

2. In each step $r = 2, \ldots, q$, $\mathcal{S}$ computes $i_j^r \leftarrow h_j(i_1^{r-1}||\ldots||i_c^{r-1}||f_{i_1^{r-1}}||\ldots||f_{i_c^{r-1}}||h(\nu, j))$. If any of the block indices i_j^r have been challenged in previous steps, $\mathcal{S}$ increments i_j^r by one (in a circular fashion in C_j) until it finds a block index that has not yet been retrieved. $\mathcal{S}$ schedules blocks $f_{i_j^r}$ for retrieval, for all $j \in \{1, \ldots, c\}$.

3. $\mathcal{S}$ sends response $R = h(f_{i_1^1}||\ldots||f_{i_c^1}||\ldots||f_{i_1^q}||\ldots||f_{i_c^q}||\nu)$ to the client, who measures the time T from the moment when challenge Q was sent.

- In Verify(G, Q, R, T), the client checks first correctness of R by recomputing the hash of all challenged blocks, and comparing the result with R. The client also checks the timing of the reply T, and accepts the response to be prompt if it falls within some specified time interval (experimental choice of time intervals within which a response is valid is dependent on drive class and is discussed in Section 6.2 below).

Security of lock-step protocol. We omit a formal analysis. Briefly, derivation of challenge values from (assumed random) block content ensures the unpredictability of challenge elements across steps in Q. $\mathcal{S}$ computes the final challenge result as a cryptographic hash of all qc file blocks retrieved in all steps. The collision-resistance of h implies that if this digest is correct, then intermediate results for all query steps are correct with overwhelming probability.

6.2 Experiments for the lock-step protocol

In this section, we perform experiments to determine the number of steps needed in the lock-step protocol to distinguish an honest server using c drives from an adversarial server employing $d < c$ drives. As discussed in Section 3.3, we evaluate both servers that "reserve" drives for RAFT testing, as well as services operating under contention.

6.2.1 Reserved drive model

We begin by looking at the "reserved" drive model, which we can test locally. The first question we attempted to answer with our tests is if we are able to distinguish an honest server from an adversarial

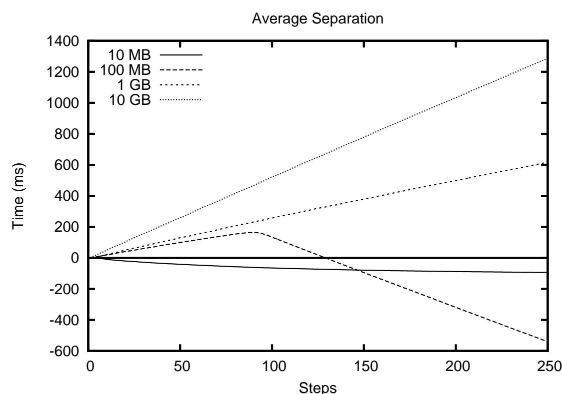

Figure 5: Average difference between adversarial and honest response times in a challenge

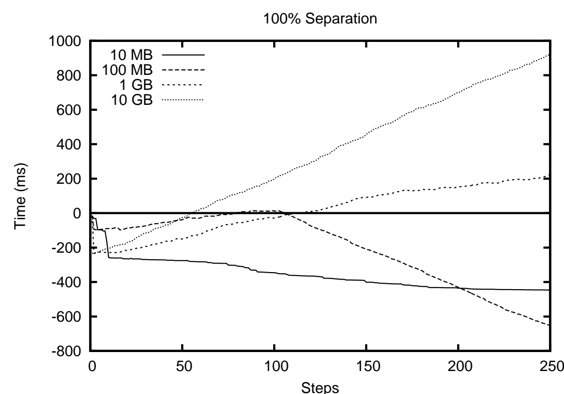

Figure 6: Complete separation of adversarial and honest response times in a challenge

one employing fewer drives based only on disk access time. Is there a range where this can be done, and how many steps in the lock-step protocol must we enforce to achieve clear separation? Intuitively, the necessity for an adversarial server employing $d \leq c - 1$ drives to read at least two blocks from a single drive in each step forces the adversary to increase its response time when the number of steps performed in the lock-step protocol increases.

Detailed experiments for $c = 3$ drives.

In practice, files are not typically distributed over a large number of drives (since this would make meta-data management difficult). Here, we focus on the practical case of $c = 3$. The file sizes presented are those of the original file before encoding and splitting across drives. Files are encoded to survive one drive failure in the honest server case and are evenly spread over the available drives. The adversarial server must store the same total amount of information in order to respond to challenges, but does so using one fewer drive than the honest server. The adversarial server spreads the encoded file evenly across two drives and must perform a double read from one of the drives in each step of the lock-step protocol.

For each file size, we performed 200 runs of the protocol for both the honest and adversarial servers. The honest server stores data on the HP, Seagate, and Fujitsu drives, while the adversary uses only the HP and Seagate drives. We show in Figure 5 the average observed difference between the adversarial and honest servers' time to reply to a challenge as a function of the number of steps in the protocol. In Figure 6, we show the 100% separation between the honest and adversarial servers defined as the difference between the minimum adversarial response time and the maximum honest response time in a challenge. Where the time in the graph is negative, an adversary using two drives could potentially convince the client that he is using three drives as promised.

One way for the adversary to cheat is to ignore the protocol entirely and instead read the whole file sequentially into memory and then respond to challenges. In the 10 MB file case this becomes the optimal strategy almost immediately, and thus both the average and complete separation graphs are entirely negative. For the 100 MB file this strategy becomes dominant around 100 steps. At this point, reading the file sequentially becomes faster than performing 100 random seeks. The turning point for the larger files is much higher and thus not visible in these graphs. Since an adversary can choose how to respond to the challenges, and may in fact read the file into memory to answer queries, a RAFT will only be effective for relatively large files.

We plot in Figures 7-10 the actual read-time histograms for both honest and adversarial servers for the shown number of steps in the lock-step protocol. Using 10 steps for the 10 MB file achieves no separation between honest and adversarial servers, due to the fact that the file fits completely in the disk buffer (running for more steps would only benefit the adversary). For a 100MB file, there is a small separation (of 14ms) between the adversarial and honest servers at 100 steps. [8] On the other hand, an honest server can respond to a 175 step challenge on a 1 GB file in roughly one second, a task which takes an adversary almost 400 ms more, and with 250 steps over a 10 GB file we can achieve nearly a second of separation between honest and adversarial servers. As discussed in Section 5.1, this degree of separation is sufficient to counteract the variability in network latency encountered in wide area networks. As such, a RAFT protocol is likely to work for files larger than 100MB when the latency between the client and cloud provider experiences little variability, and for files larger than 1GB when there is highly variable latency between the client and cloud provider.

Simulated experiments for $c > 3$ drives.

We have been comparing in all our experiments so far an honest server using $c = 3$ drives to an adversary using $d = 2$ drives. We now perform some simulations to test the effect of the number of drives the file is distributed across on the protocol's effectivness. Figure 11 shows, for different separation thresholds (given in milliseconds), the number of steps required in order to achieve 95% separation between the honest server's read times and an adversary's read times for a number of drives c ranging from 3 to 11. The honest server stores a 1 GB file, encoded for resilience to one drive failure, evenly across the available number of drives, while the adversarial server stores the same file on only $c - 1$ drives, using a balanced allocation across its drives optimized given the adversary's knowledge of Map.

The graph shows that the number of steps that need to be performed for a particular separation threshold increases linearly with the number of drives c used by the honest server. In addition, the number of steps for a fixed number of drives also increases with larger separation intervals. To distinguish between an honest server using 5 drives and an adversarial one with 4 drives at a 95% sepa-

[8] Running more than 100 steps for a 100 MB file would not benefit us here as the adversary would simply switch to a strategy of reading the entire file into memory and then answering the challenge from memory.

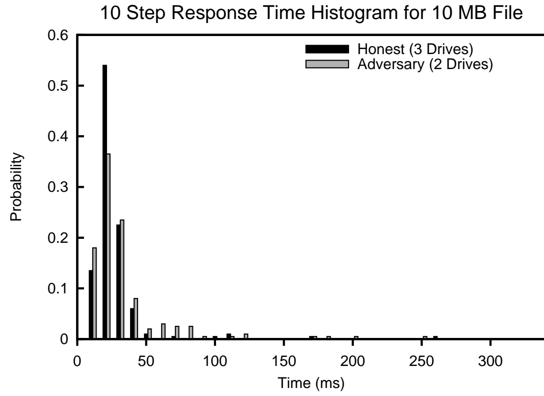

Figure 7: Read-time histogram at 10 steps for a 10 MB file

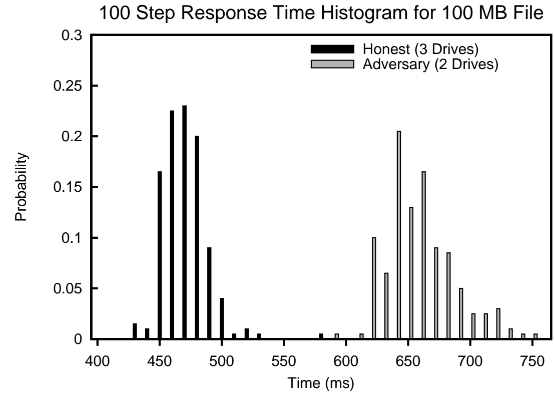

Figure 8: Read-time histogram at 100 steps for a 100 MB file

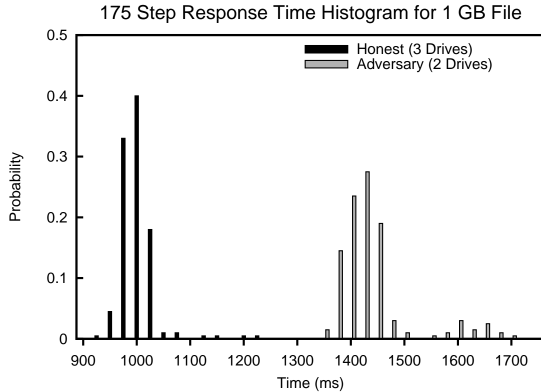

Figure 9: Read-time histogram at 175 steps for a 1 GB file

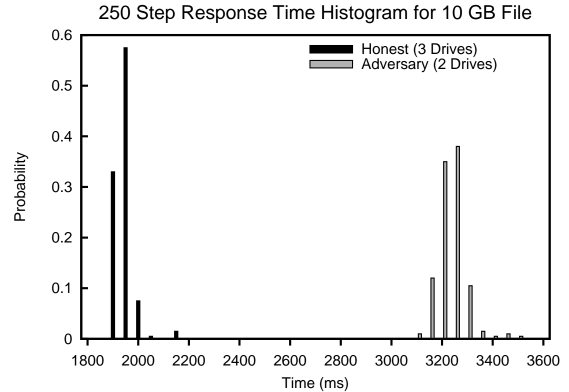

Figure 10: Read-time histogram at 250 steps for a 10 GB file

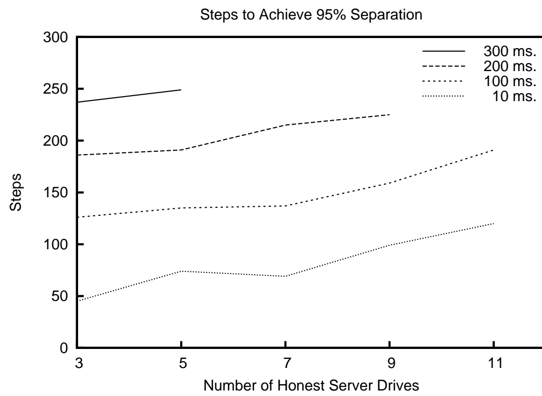

Figure 11: Effect of drives and steps on separation

ration threshold of 100ms, the lock-step protocol needs to use less than 150 steps. On the other hand, for a 300ms separation threshold, the number of steps increases to nearly 250.

More powerful adversaries.

In the experiments presented thus far we have considered an "expected" adversary, one that uses $d = c - 1$ drives, but allocates file blocks on disks evenly. Such an adversary still needs to perform a double read on at least one drive in each step of the protocol. For this adversary, we have naturally assumed that the block that is a double read in each step is stored equally likely on each of the $c-1$ drives. As such, our expected adversary has limited ability to select the drive performing a double read.

One could imagine a more powerful adversary that has some control over which drive performs a double read. As block read times are variable, the adversary would ideally like to perform the double read on the drive that completes the first block read fastest (in order to minimize its total response time). We implement such a powerful adversary by storing a full copy of the encoded file on each of the $d = c - 1$ drives available. In each step of the protocol, the adversary issues one challenge to each drive, and then the fourth challenged block to the drive that completes first.

We performed some experiments with a 2GB file. We implemented the honest server using all four of our test drives and issuing a random read to each in each step of the protocol. We then removed the OS drive (Hitachi) from the set, and implemented both the expected and the powerful adversaries with the remaining (fastest) three drives. We show in Figure 12 the average time to respond to a challenge for an honest server using $c = 4$ drives, as well as for the expected and powerful adversaries using $d = 3$ drives (the time shown includes the threading overhead needed to issue blocking read requests to multiple drives simultaneously, as well as the time to read challenged blocks from disk).

The results demonstrate that even if a powerful adversarial server is willing to store triple the necessary amount of data, it is still distinguishable from an honest server with a better than 95% probability using only a 100-step protocol. Moreover, the number of false

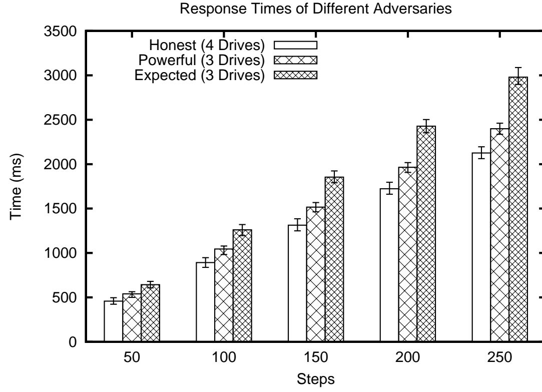

Figure 12: Time to complete lock-step protocol

negatives can be further reduced by increasing the number of steps in the protocol to achieve any desired threshold.

6.2.2 Contention model

We now turn to look at implementing RAFT in the face of contention from other users. For that, we performed tests on Mozy, a live cloud backup service. As confirmed by a system architect [19], Mozy does not use multi-tiered storage: Everything is stored in a single tier of rotational drives. Drives are not spun down and files are striped across multiple drives. An internal server addresses these drives independently and performs erasure encoding/decoding across the blocks composing file stripes. Given Mozy's use of a single tier of storage, independently addressable devices, and internal points of file-block aggregation and processing, we believe that integration of RAFT into Mozy and other similar cloud storage systems is practical and architecturally straightforward.

To demonstrate the feasibility of such integration, we performed a simple experiment. This experiment shows that even with no modification or optimization for RAFT, and in the face of contention from other users, it is possible to achieve a very basic RAFT-style demonstration that files span multiple drives.

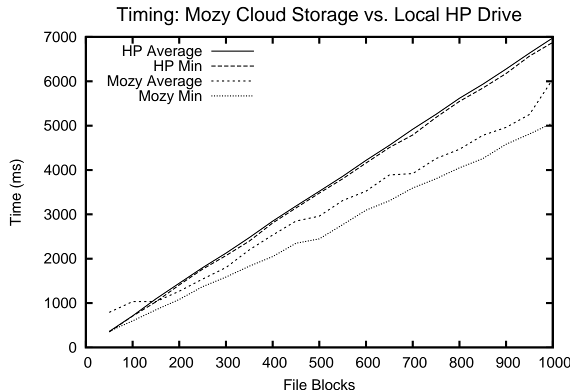

Figure 13: Comparing block retrieval on Mozy and a local drive

In this experiment, we had a remote client upload a large (64GB) file into Mozy. The client then issued several requests for randomly located bytes in the file[9]. The client measured the time required for Mozy to return the requested blocks. The time seen by the client also includes roughly 100 ms. of round-trip network latency (measured by pinging the Mozy servers). For comparison, we ran the same tests locally on the HP drive and report the results in Figure 13. Once the number of requested blocks is greater than 150, Mozy is able to retrieve the blocks and transmit them over the network faster than our local drive can pull them from disk. For example, 500 requested blocks were returned from Mozy in 2.449 seconds. By comparison, the same task took 3.451 seconds on the HP drive. Mozy consistently retrieved the blocks 15% faster (and occasionally up to 30% faster) than would be consistent with the use of a single, high-performance drive. Even with *no modification* to Mozy, we are already able to demonstrate with a RAFT-like protocol that Mozy distributes files across at least two drives.

Of course, with an integrated RAFT system in Mozy, we would expect to achieve a stronger (i.e., higher) lower bound on the number of drives in the system, along with a proof of resilience to drive crashes. RAFT's suitability for Mozy promises broader deployment opportunities in cloud infrastructure.

7. RATIONAL SERVERS

The cheap-and-lazy server model reflects the behavior of an ordinary substandard storage provider. As already noted, an efficient RAFT is not feasible for a fully malicious provider. As we now explain, though, RAFTs can support an adversarial server model that is stronger than cheap-and-lazy, but not fully Byzantine. We call such a server *rational*. We show some RAFT constructions for rational servers that are efficient, though not as practical as those for cheap-and-lazy servers.

A rational server $\mathcal{S}$ aims to constrain within some bound the drive and storage resources it devotes to file F. Refer again to Experiment $\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t)$ in Figure 1. Let $\rho(d, \{\mathcal{D}_j\}_{j=1}^d)$ be a *cost function* on a file placement $(d, \{\mathcal{D}_j\}_{j=1}^d)$ generated by $\mathcal{S}$ in this experiment. This cost function ρ may take into account d, the total number of allocated drives, and $|\mathcal{D}_j|$, the amount of storage on drive j. Let R denote an upper bound on ρ. We say that $\mathcal{S}$ is (ρ, R)-constrained if it satisfies $\rho(d, \{\mathcal{D}_j\}_{j=1}^d) \leq R$ for all block placements it generates. Roughly speaking, within constraint R, a rational server $\mathcal{S}$ seeks to maximize $\Pr[\mathsf{Acc}_{\mathcal{S}}]$. Subject to maximized $\Pr[\mathsf{Acc}_{\mathcal{S}}]$, $\mathcal{S}$ then seeks to maximize the fault-tolerance of F. Formally, we give the following definition:

DEFINITION 1. *Let p be the maximum probability $\Pr[\mathsf{Acc}_{\mathcal{S}}]$ that a (ρ, R)-constrained server $\mathcal{S}$ can possibly achieve. A (ρ, R)-constrained server $\mathcal{S}$ is* rational *if it minimizes $\Pr[\mathsf{NotFT}_{\mathcal{S}}]$ among all (ρ, R)-constrained servers $\mathcal{S}'$ with $\Pr[\mathsf{Acc}_{\mathcal{S}'}] = p$.*

A rational adversary can perform arbitrary computations over file blocks. It is more powerful than a cheap-and-lazy adversary. In fact, a rational adversary can successfully cheat against our RAFT scheme above. The following, simple example illustrates how a rational $\mathcal{S}$ can exploit erasure-code *compression*, achieving $t = 0$, i.e., no fault-tolerance, but successfully answering all challenges.

EXAMPLE 1. *Suppose that $\mathcal{S}$ aims to reduce its storage costs, i.e., minimize $\rho(d, \{\mathcal{D}_j\}_{j=1}^d) = \sum_j |\mathcal{D}_j|$. Consider a $\mathcal{RAFT}(t)$ with (systematic) encoding G, i.e., with $\{g_1, \ldots, g_m\} = \{f_1, \ldots, f_m\} = F$ and parity blocks $g_{m+1}, \ldots, g_n$. $\mathcal{S}$ can store $\{f_j\}_{j=1}^m$ individually across m disks $\{\mathcal{D}_j\}_{j=1}^m$ and* discard all parity blocks.

[9] We use bytes to ensure the requested block is on a single drive since we don't know the granularity with which Mozy stripes files.

To reply to a RAFT challenge, $\mathcal{S}$ retrieves every block of F (one per disk) and recomputes parity blocks on the fly as needed.

7.1 Incompressible erasure codes

This example illustrates why, to achieve security against rational adversaries, we introduce the concept of *incompressible* erasure codes. Intuitively, an incompressible file encoding / codeword G is such that it is infeasible for a server to compute a compact representation G'. I.e., $\mathcal{S}$ cannot feasibly compute G' such that $|G'| < |G|$ and $\mathcal{S}$ can compute any block $g_i \in G$ from G'. Viewed another way, an incompressible erasure code is one that lacks structure, e.g., linearity, that $\mathcal{S}$ can exploit to save space.[10]

Suppose that $\mathcal{S}$ is trying to create a compressed representation G' of G. Let $u = |G'| < n = |G|$ denote the length of G'. Given a bounded number of drives, a server $\mathcal{S}$ that has stored G' can, in any given timestep, access only a bounded number of file blocks / symbols of G'. We capture this resource bound by defining $r < n$ as the maximum number of symbols in G' that $\mathcal{S}$ can access to recompute any symbol / block g_i of G.

Formally, let $IEC = (\mathsf{ECEnc} : SK \times B^m \rightarrow B^n, \mathsf{ECDec} : PK \times B^n \rightarrow B^m)$ be an (n, m)-erasure code over B. Let $(sk, pk) \in (SK, PK) \leftarrow \mathsf{Keygen}(1^\ell)$ be an associated key-generation algorithm with security parameter ℓ. Let $A = (A_1, A_2^{(r)})$ be a memoryless adversary with running time polynomially bounded in ℓ. Here r denotes the maximum number of symbols / blocks that A_2 can access over G'.

Experiment $\mathsf{Exp}_A^{IEC}(m, n, \ell; u, r)$:
 $(sk, pk) \leftarrow \mathsf{Keygen}(1^\ell)$;
 $F = \{f_i\}_{i=1}^m \stackrel{R}{\leftarrow} B^m$;
 $G = \{g_i\}_{i=1}^n \leftarrow \mathsf{ECEnc}(sk, F)$;
 $G' \in B^u \leftarrow A_1(pk, G)$;
 $i \stackrel{R}{\leftarrow} Z_n$;
 $g \leftarrow A_2^{(r)}(pk, G')$;
 if $g = g_i$
  then output 1,
 else output 0

Figure 14: IEC Security Experiment

Referring to Figure 14, we have the following definition:

DEFINITION 2. *Let* $\mathsf{Adv}_A^{IEC}(m, n, \ell, u, r) = \Pr[\mathsf{Exp}_A^{IEC}(m, n, \ell; u, r) = 1] - u/n$. *We say that IEC is a (u, r)-incompressible code (for $u < n$, $r < n$) if there exists no A such that $\mathsf{Adv}_A^{IEC}(m, n, \ell; u, r)$ is non-negligible.*

In the full version of the paper [6], we prove the following theorem (as a corollary of a result on arbitrary (u, d)-incompressible IECs). It shows that given an IEC and a slightly modified query structure, a variant of our basic scheme, $\mathcal{RAFT}'(t)$, is secure against rational adversaries:

THEOREM 2. *For a system $\mathcal{RAFT}(t)$ using a $(n-1, d)$-incompressible IEC, a (ρ, R)-constrained rational adversary $\mathcal{S}$ with d drives has advantage at most $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t) \leq 1 - B(c, t, \alpha)$, where $B(c, t, \alpha)$ is defined as in Theorem 1.*

We propose two constructions for incompressible erasure codes, with various tradeoffs among security, computational efficiency, and key-management requirements:

[10]Incompressibility is loosely the inverse of local decodability [22].

Keyed RAFTs Adopting the approach of [17, 21], it is possible to encrypt the parity blocks of G (for a systematic IEC) or all of G to conceal the IEC's structure from A. (In a RAFT, the client would compute Encode, encrypting blocks individually under a symmetric key κ—in practice using, e.g., a tweakable cipher mode [18].) Under standard indistinguishability assumptions between encrypted and random blocks, this transformation implies (u, r)-incompressibility for any valid $u, r < n$. While efficient, this approach has a drawback: Fault recovery requires use of κ, i.e., client involvement.

Digital signature with message recoverability A digital signature $\sigma = \Sigma_{sk}[m]$ with message recoverability on a message m has the property that if σ verifies correctly, then m can be extracted from σ. (See, e.g., [4] for PSS-R, a popular choice based on RSA.) We conjecture that an IEC such that $g_i' = \Sigma_{sk}[g_i]$ for a message-recoverable digital signature scheme implies (u, r)-incompressibility for any valid $u, r < n$. (Formal proof of reduction to signature unforgeability is an open problem.)

This RAFT construction requires use of private key sk to compute encoding G or to reconstruct G after a data loss. Importantly, though, it doesn't require use of sk to construct F itself after a data loss. In other words, encoding is keyed, but *decoding* is keyless.

The construction is somewhat subtle. A scheme that *appends* signatures that lack message recovery does not yield an incompressible code: A can throw away parity blocks and recompute them as needed provided that it retains all signatures. Similarly, applying signatures only to parity blocks doesn't work: A can throw away message blocks and recompute them on the fly.[11]

8. CONCLUSION

We have shown how to bring a degree of transparency to the abstraction layer of cloud systems in order to reliably detect drive-failure vulnerabilities in stored files. Through theory and experimentation, we provided strong evidence that our Remote Assessment of Fault Tolerance (RAFT) works in realistic settings. With careful parametrization, a RAFT can handle the real-world challenges of network and drive operation latency for large files (at least 100MB) stored on traditional storage architectures.

We believe that RAFT has a clear deployment path in systems such as Mozy, where a cloud server can request data in parallel from multiple drives and aggregate it before communicating with the client. Such infrastructure is likely common for providers offering fault-tolerant cloud storage. As we have shown, some level of data dispersion can already be evidenced in Mozy even without server-side modification. It remains an open problem to build a full end-to-end RAFT protocol integrated into an existing cloud infrastructure and demonstrate its feasibility in practice.

With their unusual combination of coding theory, cryptography, and hardware profiling, we feel that RAFTs offer an intriguing new slant on system assurance. RAFT design also prompts interesting new research questions, such as the modeling of adversaries in cloud storage systems, the construction of provable and efficient incompressible erasure codes, and so forth.

[11]Message-recoverable signatures are longer than their associated messages. An open problem is whether, for random F, there is some good message-recoverable signature scheme over blocks of G that has no message expansion. Signatures would be existentially forgeable, but checkable against the client copy of F.

Acknowledgments

We wish to extend our thanks to Burt Kaliski for his comments on an early draft of this paper, and to Erik Riedel for clearing up questions about hard drive operation. We also thank Tom Ristenpart for shepherding the paper and to the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] The Hadoop distributed file system. `http://hadoop.apache.org/hdfs`.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. ACM SOSP*, 2009.

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

[4] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In U. Maurer, editor, *Proc. EUROCRYPT '96*, volume 1070 of *LNCS*, pages 399–416. Springer-Verlag, 1989.

[5] K. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proc. ACM CCS '09*, pages 187–198, 2009.

[6] K. D. Bowers, M. van Dijk, A. Juels, A Oprea, and R. Rivest. How to tell if your cloud files are vulnerable to drive crashes, 2010. IACR ePrint manuscript 2010/214.

[7] S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In *Proc. EUROCRYPT '93*, pages 344–359. Springer, 1993. LNCS vol. 765.

[8] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *Proc. USENIX Annula Technical Conference*, 2009.

[9] J. Cox. T-Mobile, Microsoft tell Sidekick users we 'continue to do all we can' to restore data. *Network World*, October 13, 2009.

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

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

[12] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E.F. Brickell, editor, *Proc. CRYPTO '92*, pages 139–147. Springer, 1992. LNCS vol. 740.

[13] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proc. CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proc. ACM SOSP*, pages 193–206, 2003.

[15] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. In *Proc. ACM SOSP*, pages 29–43, 2003.

[16] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Proc. Financial Cryptography*, pages 120–135. Springer, 2002. LNCS vol. 2357.

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

[18] S. Halevi and P. Rogaway. A tweakable enciphering mode. In D. Boneh, editor, *Proc. CRYPTO'03*, volume 2729 of *LNCS*, pages 482–499. Springer, 2003.

[19] Mozy CTO J. Herlocker. Personal Communication, 2011.

[20] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proc. ISOC NDSS*, pages 151–165, 1999.

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

[22] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proc. STOC*, pages 80–86, 2000.

[23] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: a durable and practical storage system. In *Proc. USENIX'07*, pages 10:1–10:14, Berkeley, CA, USA, 2007. USENIX Association.

[24] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee. Triangle inequality variations in the internet. In *Proc. ACM IMC*, 2009.

[25] R. Merkle. A certified digital signature. In *Proc. Crypto 1989*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[26] J.F. Muth and G.L. Thompson. *Industrial scheduling*. Prentice-Hall, 1963.

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

[28] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in dram. *SIGOPS Operating Systems Review*, 43(4):92–105.

[29] E. Riedel, C. Van Ingen, and J. Gray. A performance study of sequential I/O on Windows NT 4.0. Technical Report MSR-TR-97-34, Microsoft Research, September 1997.

[30] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[31] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proc. Asiacrypt 2008*, volume 5350 of *LNCS*, pages 90–107. Springer-Verlag, 2008.

[32] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proc. ACM Sigmetrics*, pages 241–251, 1994.