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

Kevin D. Bowers, Marten van Dijk, Ari Juels, Alina Oprea
RSA Laboratories
Cambridge, MA, USA
(kbowers, marten.vandijk, ajuels, 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 show experimentally that RAFTs can work in practice.**

I. 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 with redundancy or error correction to protect against data loss. Amazon, for example, claims that its S3 service stores three replicas of each file[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., [5], 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 each of 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 and Figures 1 and 2 give 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 service, "Cheapskate Pizza." Recently, however, Cheapskate failed to deliver pizzas at all for the big pregame party. Cheapskate said that their only pizza oven had suffered a catastrophic failure, but that they are 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

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

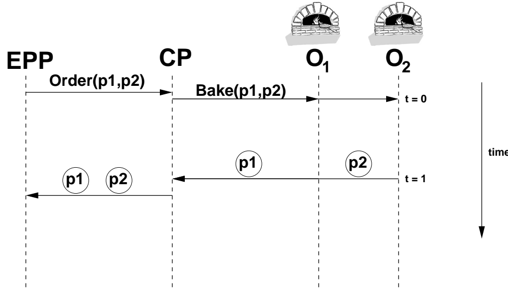

Figure 1: Resilient pizza shop with two ovens

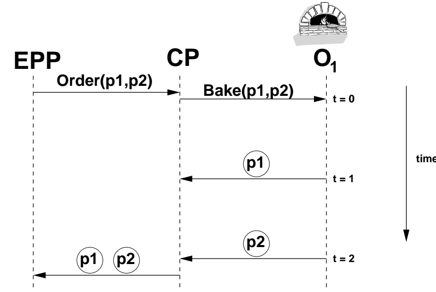

Figure 2: Failure-prone pizza shop with only one oven

> minutes, he places an order for two dozen pizzas, for delivery to the frat 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, so that Cheapskate wouldn't be able just to deliver warmed up pre-made pizzas.
>
> Cheapskate is only a fifteen minute drive from the frat. So 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 frat'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 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 also empirically explore another real-world complication, variance in network latencies between remotely located clients and servers. Despite these design challenges, we propose and experimentally validate a RAFT that can, for example, distinguish sharply between a four-drive system with fault tolerance $t = 1$ and a three-drive system with no fault tolerance in less than 500ms.

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.

[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.)

A. Related work

Proofs of Data Possession (PDPs) [1] and Proofs of Retrievability (PORs) [6], [7], [15], [23] 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 [20]. Both [17] and [3] 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 [8]. 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 [14]. Our protocols here measure not computational resources, but the storage resources devoted to a file. (Less directly related is physical distance bounding, introduced in a cryptographic setting in [4]. 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, [1], [11] 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., [10]. 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.

B. Organization

Section II gives an overview of key ideas and techniques in our RAFT scheme. We present formal adversarial and system models in Section III. In Section IV, we introduce a basic RAFT in a simple system model. Drawing on experiments, we refine this system model in Section V, resulting in a more sophisticated RAFT in Section VI. In Section VII, we formalize an economically rational adversarial model and sketch matching RAFT constructions. We conclude in Section VIII with discussion of future directions.

II. Overview: Crafting a RAFT

We now discuss in further detail the practical technical challenges in crafting 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 on a drive 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. We examine representative Internet routes experimentally and find that network latency is, for our purposes, quite stable; it has little impact on RAFT design.

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 [9] for generating q independent, unpredictable sets of challenges non-interactively. (File blocks serve as a kind of "commitment.") The server's response to Q then is simply the aggregate (hash) of all of the cq file blocks it accesses.

III. 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 client first encodes the file by adding some redundancy. 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.

A. 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:

- $\mathsf{Keygen}(1^\ell) \overset{R}{\rightarrow} \kappa$: 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 by the client to an m-block file $F = \{f_i\}_{i=1}^m$; it takes additional inputs 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, or unkeyed, e.g., applying an erasure code or keyless cryptographic operation to F.
- $\mathsf{Map}(n, t, c) \rightarrow \{C_j\}_{j=1}^c$: 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 $\perp$. (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. 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 $\mathsf{Reconstruct}$ 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 $\mathcal{RAFT}(t) = \{\mathsf{Keygen}, \mathsf{Encode}, \mathsf{Map}, \mathsf{Challenge}, \mathsf{Response}, \mathsf{Verify}, \mathsf{Reconstruct}\}$.

B. 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, and the placement is established entirely by the server. In this paper, we only consider layout-specified protocols, although layout-free protocols are an interesting point in the RAFT design space worth exploring.

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., [18], to a model in which the file is maintained only by the provider and the client deletes the local copy after outsourcing the file.

C. 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 [22]. 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].

Modeling network latency: We assume in our model that we can accurately estimate the network latency between the client and the server. We will present some experimental data on network latencies and adapt our protocols to handle observed small variations in time.

D. 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. Before sending F to $\mathcal{S}$, the client applies Encode, yielding an encoded file G of size n.

Both the client and server can compute the logical placement $\{C_j\}_{j=1}^{c}$ by applying the Map function. The server 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.

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 files 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 prior to receiving 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. When it receives G, server $\mathcal{S}$ applies this mapping to the n constituent blocks.

General adversarial model: It is also useful to consider a general adversarial model, cast in an experimental

[3]At the time of this writing, SSDs are still considerable more expensive than rotational drives and have nowhere near the capacity. A typical rotational drive can be bought for roughly \$0.07/GB in capacities up to 2 TBs, while most SSDs still cost more than \$2.00/GB are are only a few hundred GBs in size. For a 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.

framework. We define the security of our system $\mathcal{RAFT}(t)$ according to the experiment from Figure 3. 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 $\mathsf{Exp}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t)$:
  $\kappa \leftarrow \mathsf{Keygen}(1^\ell)$;
  $F = \{f_i\}_{i=1}^m \leftarrow_R B^m$;
  $G = \{g_i\}_{i=1}^n \leftarrow \mathsf{Encode}(\kappa, F, t, c)$;
  $(d, \{\mathcal{D}_j\}_{j=1}^d) \leftarrow \mathcal{S}^{\mathcal{O}(\kappa)}(n, G, t, c, \text{“store file”})$;
  $Q \leftarrow \mathsf{Challenge}(n, G, t, c)$;
  $(R, T) \leftarrow \mathcal{S}^{\{\mathcal{D}_j\}_{j=1}^d}(Q, \text{“compute response”})$;
  if $\mathsf{Acc}_{\mathcal{S}}$ and $\mathsf{NotFT}_{\mathcal{S}}$
    then output 1,
  else output 0

Figure 3: 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., that 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 VII, 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]

E. Problem Instances

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

IV. 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.

A. 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 client then spreads the encoded file blocks evenly on c drives and specifies the layout. To determine that the server respects the 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

[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.

$(\mathsf{ECEnc}, \mathsf{ECDec})$ 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 $j \in \{1, \ldots, c\}$.
- $\mathsf{Response}(Q)$ 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. 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 $r < m$.

B. Security Analysis

We start by computing $\Pr[\mathsf{Acc}_{\mathcal{S}}]$ and $\Pr[\mathsf{NotFT}_{\mathcal{S}}]$ for an honest server. Then we turn our attention to bounding the advantage of a cheap-and-lazy server.

Lemma 1: For an honest server, $\Pr[\mathsf{Acc}_{\mathcal{S}}] = 1$.

Proof: An honest server respects the layout specified by the client, and can answer all queries in time $\tau_{\ell} + L$. ■

Lemma 2: If $\alpha \geq t/(c-t)$, then $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$ for an honest server.

Proof: An honest server stores the file blocks to disks deterministically, as specified by the protocol. Then any set of t drives stores tn/c blocks. From the restriction on α, it follows that any t drives store at most αm blocks, and thus the file can be recovered from the remaining $c-t$ drives. ■

Lemma 3: Denote by $\epsilon_{\mathcal{S}}$ the "double-read" probability that the cheap-and-lazy server $\mathcal{S}$ cannot answer a c-block challenge by performing single block reads across all its drives. For fixed c, t and α with $\alpha \geq t/(c-t)$, if

$$\epsilon_{\mathcal{S}} \leq B(c, t, \alpha) = \frac{\alpha(c-t) - t}{(1+\alpha)(c-t)},$$

then $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

Our timing assumptions (constant network latency and constant block read time) imply that $\epsilon_{\mathcal{S}}$ is equal to the probability that the server does not answer c-block challenges correctly and promptly, i.e., $\epsilon_{\mathcal{S}} = 1 - \Pr[\mathsf{Acc}_{\mathcal{S}}]$.

Proof: The server might use a different placement than the one specified by Map; the placement can be unbalanced, and encoded file blocks can be duplicated or not stored at all. Our goal is to show that the server stores a sufficient number of blocks on any $d-t$ drives. Let X be a fixed set of t out of the d server drives. Let Y be the remaining $d-t$ drives and denote by δ the fraction of encoded file blocks stored on drives in Y. We compute a lower bound on δ.

For a query Q consisting of c encoded file block indices, let δ_Q denote the fraction of encoded file blocks indexed by Q stored on drives in Y. Let $\mathcal{Q}$ be the query space consisting of queries computed by Challenge. Since every block index is covered an equal number of times by queries in $\mathcal{Q}$, if we consider δ_Q a random variable with all queries in $\mathcal{Q}$ equally likely, it follows that $E[\delta_Q] = \delta$.

Consider a query Q randomly chosen from $\mathcal{Q}$. With probability $1 - \epsilon_{\mathcal{S}}$, the server can correctly answer query Q by performing single reads across all its drives. Since $\mathcal{S}$ satisfies the block obliviousness and block atomicity assumptions, $\mathcal{S}$ does not computationally process read blocks. This means that at most t out of the c blocks indexed by Q are stored on drives in X. Or, equivalently, at least $c-t$ encoded file blocks indexed by Q are stored on drives in Y. We can infer that for queries that succeed, $\delta_Q \geq (c-t)/c$. This yields the lower bound $\delta = E[\delta_Q] \geq (1 - \epsilon_{\mathcal{S}})(c-t)/c$.

From the bound on $\epsilon_{\mathcal{S}}$ assumed in the statement, we obtain that $\delta n \geq m$. From the block obliviousness and block atomicity assumptions for $\mathcal{S}$, it follows that drives in Y store at least m encoded file blocks. We can now easily infer t-fault tolerance: if drives in X fail, file F can be reconstructed from blocks stored on drives in Y, for any such Y via ECDec. ■

Based on the above lemmas, we prove the theorem:

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, $\mathsf{Adv}_{\mathcal{S}}^{\mathcal{RAFT}(t)}(m, \ell, t) \leq 1 - B(c, t, \alpha)$.

Proof:

1. Completeness follows from Lemmas 1 and 2.

2. If $d < c$, assume that $\mathcal{S}$ needs to answer query Q consisting of c blocks. Then $\mathcal{S}$ has to read at least two blocks in Q from the same drive. This results in a timing of the response of at least $2\tau_{\ell} + \mathsf{L}$. Thus, $\mathcal{S}$ is not able to pass the verification algorithm and its advantage is 0.

3. Lemma 3 states that $\mathcal{S}$ satisfies $1 - \Pr[\mathsf{Acc}_{\mathcal{S}}] = \epsilon_{\mathcal{S}} > B(c, t, \alpha)$ or $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$. So, the server's advantage, which is defined as $\Pr[\mathsf{Acc}_{\mathcal{S}}$ and $\mathsf{NotFT}_{\mathcal{S}}]$, is at most the minimum $\min\{\Pr[\mathsf{Acc}_{\mathcal{S}}], \Pr[\mathsf{NotFT}_{\mathcal{S}}]\} \leq 1 - B(c, t, \alpha)$. ■

Multiple-step protocols: We can make use of standard probability amplification techniques to further reduce the advantage of a server. For example, we 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.

V. Network and drive timing model

In the simple model of Section IV, 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 let us effectively treat network latency as constant and block-read time as a fixed probability distribution. We also show how to cast our RAFT protocol as a statistical hypothesis testing problem, a view that aids analysis of our practical protocol construction in the next section.

A. Network model

We present some experimental data on network latency between hosts in different geographical locations, and conclude that it exhibits fairly limited variance over time. We discuss approaches to factoring this variance out of our protocol design. We also show how to reduce the communication complexity of our protocol—thereby eliminating network-timing variance due to fluctuations in network bandwidth.

Network latency model: To measure network latency over time, we ping 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. Figure 4 shows the ping times for the one week interval, as well as cutoffs for various percentages of the data. The y-axis is presented in log-scale.

The ping times to Santa Clara during the week in question ranged from 86 ms to 463 ms. While only 0.1% of measured ping times come in at the minimum, 65.4% of the readings were 87 ms, and 93.4% of readings were 88 ms or less. The ping-time distribution is clearly heavy tailed, with 99% of ping times ranging within 10% of the average. Moreover, spikes in ping times are correlated in time, and are most likely due to temporary network congestion.

Ping times to Shanghai exhibit more variability across the larger geographical distance. Round-trip ping times range between 262 ms and 724 ms. The ping time distribution is also heavy tailed, however, with spikes correlated in time. 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.

We offer several possible strategies to factor the variability in network latency out of our protocol design. A simple approach is to abort the protocol in the few cases where the response time of $\mathcal{S}$ exceeds 110% of the average response time (or another predetermined threshold). The protocol could be restarted at a later, unpredictable time. A different approach exploits the temporal and geographical correlation of ping times. The idea is to estimate network latency to a particular host by pinging a trusted machine located nearby geographically. Such measurement of network congestion would determine whether the route in question is in a stable interval or an interval of ping-time spiking. To test this idea, we performed an experiment over the course of a week in April 2010, pinging two machines in Santa Clara, and two machines in Shanghai simultaneously from our Boston machine. We observed that, with rare exceptions (which we believe to be machine dependent), response times of machines in the same location (Santa Clara or Shanghai) exhibited nearly identical ping times.

A more general and robust approach, given a trusted measurement of the network latency over time between two hosts, is to consider a response valid if it arrives within the maximum characterized network latency. We could then adopt the bounding assumption that the difference between minimum and maximum network latency (377 ms for Santa Clara, and 462ms for Shanghai) 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. We quantify the effect of such file-block prefetching in Appendix A.

Limited network bandwidth: In the basic protocol from Section IV, 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.

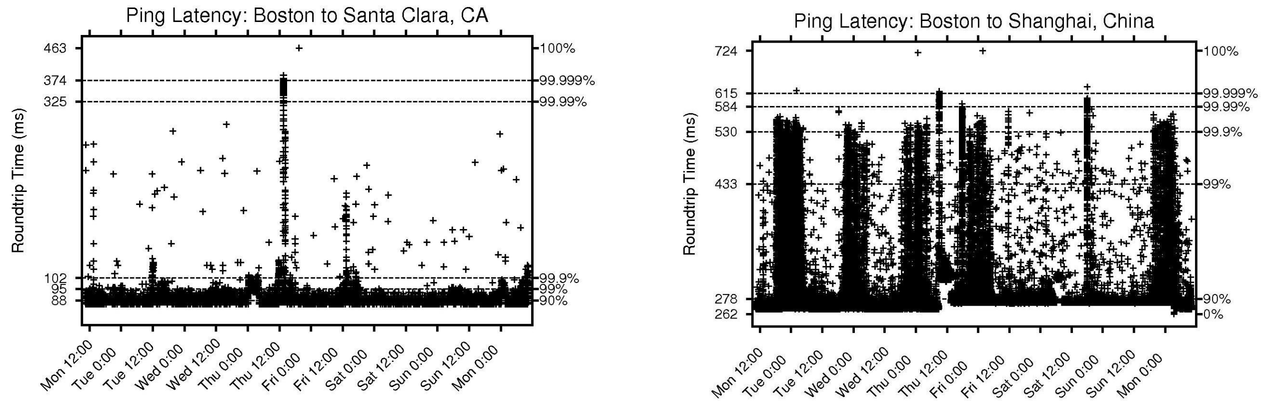

Figure 4: Ping times between Boston, MA and Santa Clara, CA (left), and between Boston, MA and Shanghai, China (right)

B. 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 15000 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 on the desired sector, the actual data transfer is performed. The data transfer rate (or throughput) depends on several factors, but is on the order of 100MB per second for high-end drives today. The disk controller maintains an internal cache and implements some 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 [22].

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[6], and Seagate. Complete specifications for each drive can be found in Table I.

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 [21] 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 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.

As Figure 5 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 [21] that the OS issues requests to disks for blocks of logical size 64KB, and there is no noticeable difference

[6]Upon further inspection, the HP drive is actually manufactured by Seagate. Several brands of drives available today are in fact made by one of the three manufacturers identified in this test

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

Table I: Drive specifications

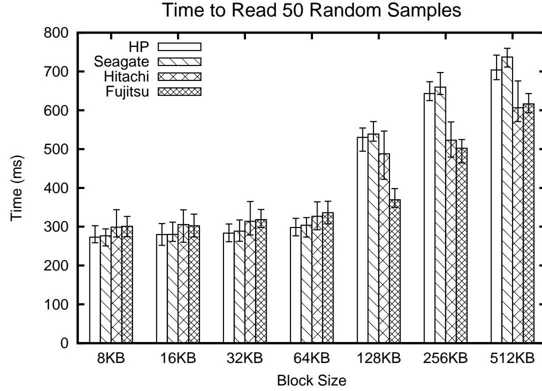

Figure 5: Read time for 50 random blocks

in the time to read blocks up to 64KB.

For our purposes, therefore, a remote server can read 64KB random block 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 6 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.

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%.

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 6, 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. Later, in Section VI, we modify our RAFT to smooth out seek-time variance. The idea is to sample (seek) many blocks in succession.

C. Statistical framework

The nub of our RAFT is that the client tries, by measuring the timing of a response, to distinguish between an honest server and a cheap-and-lazy adversary who may not be t fault-tolerant. It is helpful to model this measurement as a statistical hypothesis testing problem for the client. The client sends a challenge Q and measures the timing of the server's response T.

Let D_s be a random variable denoting the time for the server to answer a query by performing single block reads across its drives. Let D_d be the time for the server to answer a query by performing a double block read on at least one drive. Let $1-\epsilon$ be the probability that the server can answer a query with only single block reads on its drives. Then $D(\epsilon) = (1-\epsilon)D_s + \epsilon D_d$ is the random variable representing the disk access time to answer a random query.

For constant network latency L, the timing of a response follows a distribution $T(\epsilon) = D(\epsilon) + \mathsf{L}$. For an honest server the timing is distributed according to $T(\epsilon = 0)$. Let $B = B(c, t, \alpha)$ be the bound on ϵ in Lemma 3; only for $\epsilon \leq B$ is a cheap-and-lazy server guaranteed to be t fault-tolerant. So, the client needs to distinguish whether the timing of the server's response is an instance of $T(\epsilon = 0)$ (honest server) or $T(\epsilon > B)$ (a cheap-and-lazy server who may not be t fault-tolerant).

In this statistical hypothesis-testing view, a false positive occurs when the client labels an honest server as adversarial; the completeness of the protocol is the probability that this misclassification doesn't take place. A false negative occurs when the client fails to detect an adversarial server who is not t fault-tolerant; this probability is the adversarial advantage.

In the practical RAFT protocol we consider below, we craft a query Q such that the server must read multiple blocks in succession in order to return its response. Thus, the response time T is an aggregate across many reads, i.e.,

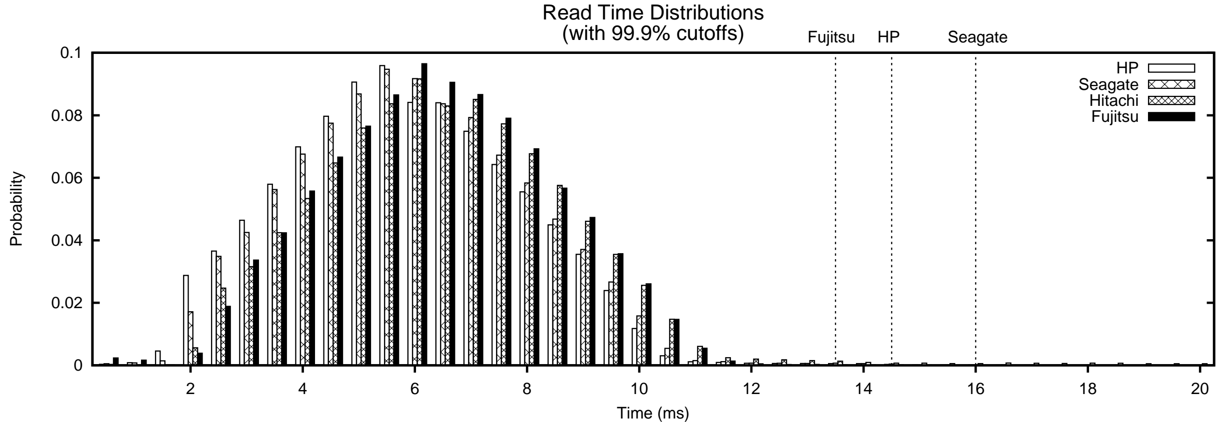

Figure 6: Read time distribution for 64KB blocks

the sum of multiple, independent instances of $D(\epsilon)$ (plus L). The effect of summing independent, identically distributed random variables in this way is to lower the variance of T. Reducing the variance of T helps the client separate the timing distribution of an honest server from that of a cheap-and-lazy server who may not be t fault-tolerant.

Remark: If the network latency L is variable, then an adversary may feign a high network latency L_A in order to fetch additional blocks. In this regime, we have a new bound $B_A(c,t,\alpha)$; see Appendix A. The client then needs to distinguish $T(\epsilon = 0)$ from $T_A(\epsilon > B_A) = D(\epsilon > B_A) + \mathsf{L}_A$.

VI. Practical RAFT Protocol

In this section, we propose a practical variant of the basic RAFT protocol from Section IV. As discussed in Section V the main challenge in practical settings is the high variability in drive seek time. (While network latency also exhibits some variability, recall that in Section V-A we proposed a few approaches to handle occasional spikes.) 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 IV. We show experimentally that with enough steps, the client can with high probability distinguish between a correct server and an adversarial one in our statistical framework of Section V-C. We also discuss practical parameter settings and erasure-coding choices.

A. 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 [19], 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 [25]. 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 [9] 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 $\mathsf{Reconstruct}$ algorithms similar to our basic RAFT. 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 collision-resistant hash functions that output indices in C_j: $h_j \in \{0,1\}^* \rightarrow C_j$. Let h be a cryptographically secure hash function with fixed

output (e.g., from the SHA family).

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

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

- Algorithm $\mathsf{Response}(Q)$ consists of the following steps:

1. $\mathcal{S}$ reads file blocks $f_{i_1^1}, \ldots, f_{i_c^1}$ in Q.
2. In each step $r = 2, \ldots, q$, $\mathcal{S}$ computes $i_j^r \leftarrow h_j(i_1^{r-1}|| \ldots ||i_c^{r-1}||f_{i_1^{r-1}}|| \ldots ||f_{i_c^{r-1}})$. 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 to the client response $R = h(f_{i_1^1}|| \ldots ||f_{i_c^1}|| \ldots ||f_{i_1^q}|| \ldots ||f_{i_c^q})$ and the client measures time T from the moment when challenge Q was sent.

- In $\mathsf{Verify}(G, Q, R, T)$, 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 characteristics and is discussed in Section VI-B 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.

B. 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 $d \leq c - 1$ drives. We show results for a number of drives c ranging from 2 to 20, as well as detailed experimental data on disk access distribution for honest and adversarial servers for $c = 4$ drives.

As described in Section V-B, we collected 100,000 samples of 64KB block reads randomly chosen from a 2GB file stored on each of our test drives to generate the distributions in Figure 6. From this data, we now simulate experiments on multiple homogenous drives. To give the adversary as much advantage as possible, we compare an adversary using only HP drives (fastest average response times) against an honest server using only Fujitsu drives (slowest average response times).

In our tests, read time is primarily affected by the distance the read head has to move. Our test files are all 2GB in size (relatively small compared to the drive's capacity), and thus there is limited head movement between random samples. The first read in any test, however, results in a potentially large read-head move as the head may be positioned anywhere on the disk. We account for this and other order-sensitive results in all our simulated tests by simulating a drive using a randomly selected complete run from our previous tests. There are 400 such runs, and thus 400 potential "drives" to select in our tests. Each test is repeated 500 times, selecting different drives each time.

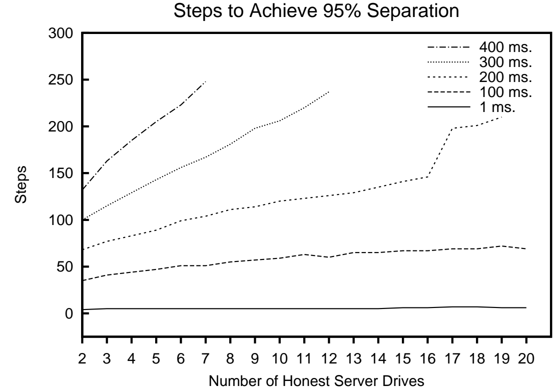

Figure 7: Effect of drives and steps on separation

Number of steps to ensure timing separation: The first question we attempted to answer with our simulation is if we are able to distinguish an honest server from an adversarial 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.

Figure 7 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. The honest server stores a 2GB file fragment on each of its c drives, while the adversarial server stores the 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 4 drives and an adversarial one with 3 drives at a 95% separation threshold of 100ms, the lock-step protocol needs to use less than 50 steps. On the other hand, for a 400ms separation threshold,

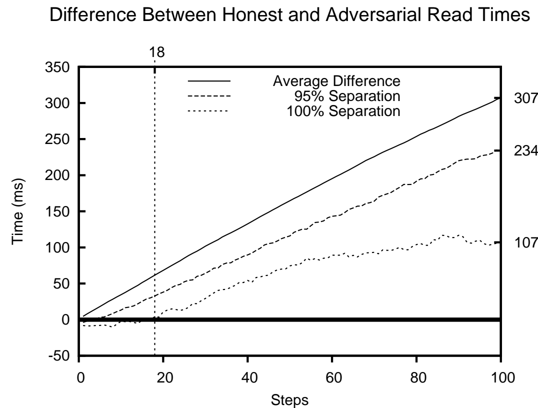

Figure 8: Separation as number of steps increases

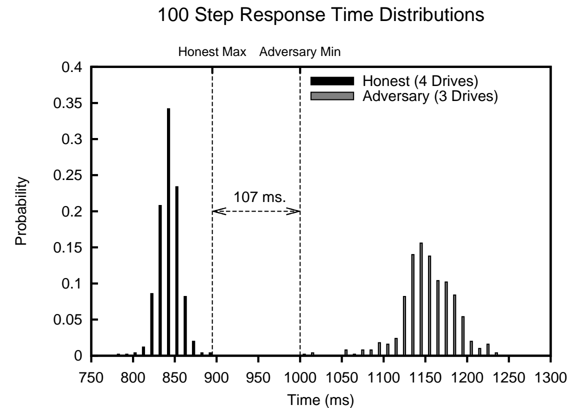

Figure 9: Distribution of 500 simulated runs

the number of steps increases to 185.

The sharp rise in the 200 ms. line at 17 drives in Figure 7 is due to the outlying 132 ms. read present in the Fujitsu data. That read occurred at step 193 in one of the 400 runs. With 17 drives, that run was selected in more than 5% of the 500 simulated tests, and so drove up the 95th percentile difference between honest and adversarial. No such outlier was recorded in the 100,000 samples taken from the HP drive, and the affect is not visible in the 300 and 400 ms. lines due to the lower number of drives.

Detailed experiments for $c = 4$ drives: In practice, files are not typically distributed on a large number of drives (since this would make meta-data management difficult). In the remaining part of this section, we focus on the practical case of $c = 4$ drives and provide more detailed experimental data on read time distributions for both honest and adversarial servers.

For this case, it is worth further analyzing the effect of the number of steps in the protocol on the separation achieved between the honest and adversarial server. Figure 8 shows the difference between the adversarial read time and the honest server read time (computed over 500 runs). Where the time in the graph is negative, an adversary using fewer than 4 drives could potentially convince the client that he is using 4 drives. The average adversarial read time is always higher than the average honest read time, but there does exist some overlap in the read access distributions where false positives are possible for protocols shorter than 18 steps. Above that point, the two distributions do not overlap, and in fact using only 100 steps we are able to achieve a separation of 107 ms. between the honest server's worst run, and the adversary's best.

We plot in Figure 9 the actual read time distributions for both honest and adversarial servers for 100 steps in the lock-step protocol. We notice that the average response time for the honest server is at around 840 ms, while the average response time for the adversarial server is close to 1150ms (a difference of 310 ms, as expected).

More powerful adversaries: In the experiments done so 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 $c - 1$ server 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 the drive performing 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 show in Figure 10 average response times for an honest server with $c = 4$ drives, as well as two adversarial servers using $d = 3$ drives. The adversaries, from right to left, are the expected adversary and the adversary for which double-reads always take place on the drive that first completes the first block read in each step of the protocol. In order for the adversary to choose which drive performs the double read the entire file must be stored on each drive. Otherwise the adversary cannot wait until the first read completes and then choose the drive that returns first to perform the double read, as the block necessary to perform the double read may not be available on that drive. Thus, although this adversary is achievable in practice, in the $c = 4$ case, he requires three times the storage of an honest server.

The bars in the graph indicate the average read times for the given number of steps, with the error bars indicating the 5th and 95th percentile of the data, as collected over 100 actual runs of the protocol. To model the honest server, we used all 4 of our test drives, issuing a random read to each in each step of the protocol. We then removed the OS drive (Hitachi) from the set, and repeated the tests as the adversary, either allowing it to select the fastest drive to

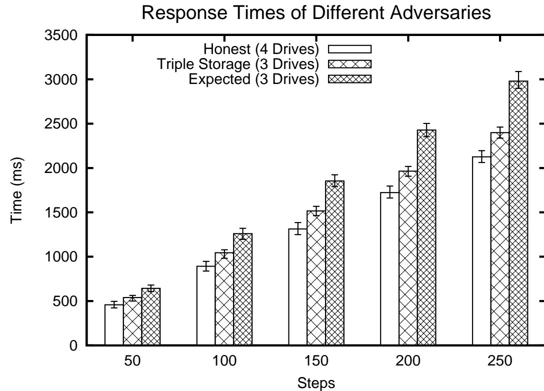

Figure 10: Time to complete lock-step protocol

perform the double read, or randomly selecting the double read drive in each round. This graph thus indicates the actual time necessary to respond to the protocol, including the threading overhead necessary to issue blocking read requests to multiple drives simultaneously.

Figure 10 shows that even if an adversarial $\mathcal{S}$ is willing to store triple the necessary amount of data, it is still distinguishable from an honest $\mathcal{S}$ with a better than 95% probability using only a 100-step protocol. Moreover, the number of false negatives can be further reduced by increasing the number of steps in the protocol to achieve any desired threshold.

C. Parameters for detecting more powerful adversaries

As we have seen in the previous section, the client can efficiently distinguish between an honest server and an adversarial server using $d \leq c - 1$ drives. For example, with $c = 4$, the lock-step protocol needs about 100 steps to create a 250 ms. separation between the expected adversarial server and an honest server with 95% probability, taking on average 900ms in disk access time). Here we discuss the additional protocol cost incurred for detecting more sophisticated adversaries, in particular adversaries that use $d \geq c$ drives, but are not fault-tolerant (by employing an uneven placement, for instance).

Figure 9 shows that, for $c = 4$, the aggregate response time of 100 steps of single reads across each drive is distributed with mean $\approx$ 850 ms. and standard deviation $\approx$ 14 ms, and the aggregate response time of 100 steps of at least one drive performing a double read is distributed with mean $\approx$ 1150 ms. and standard deviation $\approx$ 31 ms. So, for double-read probability ϵ, the aggregate response time of 100 steps, denoted by random variable $T_{100}(\epsilon)$, has mean $E[T_{100}(\epsilon)] \approx (1-\epsilon) \cdot 850 + \epsilon \cdot 1150 = 850 + \epsilon \cdot 300$ and deviation $\sigma(T_{100}(\epsilon)) \approx \sqrt{196 + \epsilon \cdot 765}$.

Suppose that we want to tolerate $t = 1$ drive failures, and suppose that we allow an expansion rate $1 + \alpha = 1.75$. For these parameters the bound in Lemma 3 yields $B(5, 1, 0.75) = 2/7$. As explained in Section V-C, we want to distinguish whether $\epsilon = 0$ or $\epsilon \geq 2/7$ in order to determine if the server is 1 fault-tolerant. Suppose we use $q = 100 \cdot s$ steps in our lock-step protocol. Then $E[T_{100s}(\epsilon)] \approx (850 + 300\epsilon) \cdot s$ and $\sigma(T_{100s}(\epsilon)) \approx \sqrt{(196 + \epsilon \cdot 765)s}$. In order to separate $T_{100s}(\epsilon = 0)$ from $T_{100s}(\epsilon \geq 2/7)$, we could separate their means by 4 standard deviations such that their tails cross at about 2 standard deviations from each mean. By the law of large numbers, the aggregate response time of $100 \cdot s$ steps is approximated by a normal distribution. Since the cumulative probability of a normal distributed tail starting at 2σ has probability ≈ 0.022, both the false positive and false negative rates are $\approx 2.2\%$.

Requiring 4 standard deviations separation yields the inequality $300 \cdot 2/7 \cdot s \geq 2 \cdot 34.4 \cdot \sqrt{s}$, that is $s \geq 0.643$. We need $q = 64$ steps, which take about 650 ms. disk access time. Given c, t, and bounds on the false positive and negative rates, the example shows a tradeoff between the number of steps (q) in the challenge-response protocol and additional storage overhead (α); the lock-step protocol is faster for an erasure code with higher rate.

D. Efficient Erasure Coding

We have discussed parameter choices required for the lock-step protocol to distinguish honest servers from adversarial ones. Here we briefly discuss the practicality of the encoding algorithm in our RAFT protocol. The most practical erasure codes in use today are Raptor codes [24]. We performed some experiments with a licensed Raptor code package from Digital Fountain. We observe encoding speeds on the order of several dozen seconds for a 2GB file, using a systematic encoding. (Given the absence of officially published performance figures, we avoid giving precise ones here.) In constrast, whole-file Reed-Solomon encoding is not practical for files of that size.

With the current implementation of Raptor codes, there are some restrictions on symbol sizes and message sizes that can be encoded at once. For 64KB symbol size, we are able to encode files of size up to 3.4GB (we have used a 2GB file in our experiments). The code rate α needs to satisfy $\alpha \geq t/(c-t)$, from our security analysis (Theorem 1). For instance, for a placement using $c = 4$ drives, tolerating $t = 1$ failures, this implies that the code rate should be at least 0.33. Raptor codes have the advantage of being able to generate an arbitrarily large number of parity blocks. We can generate an encoding with rate 30% at half of the throughput of generating an encoding with rate 1%.

VII. 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 3. 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*. 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.

A. 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.[7]

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

Figure 11: IEC Security Experiment

Referring to Figure 11, 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 Appendix A, we prove the following theorem (as a corollary of a result on arbitrary (u, d)-incompressible IECs). It shows that with a slightly modified query structure and given an IEC, a variant $\mathcal{RAFT}'(t)$ of our basic scheme is secure against rational adversaries:

Theorem 2: For $\mathcal{RAFT}'(t)$ using a $(n-1, d)$-incompressible IEC, if a (ρ, R)-constrained rational adversary $\mathcal{S}$ with d drives has double-read probability $\epsilon \leq B(c, t, \alpha)$, then $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

B. Incompressible erasure-code constructions

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

Keyed RAFTs Adopting the approach of [12], [15], 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,

[7]Incompressibility is loosely the inverse of local decodability [16].

e.g., a tweakable cipher mode [13].) 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., [2] 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.[8]

We leave as an intriguing open problem the incompressibility of MDS erasure codes, is it possible to beat the bound of Appendix B? We also leave as an open problem the question of incompressibility for practical erasure codes such as Raptor codes.

VIII. 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 demonstrated the effectiveness of our Remote Assessment of Fault Tolerance (RAFT), a scheme that tests fault tolerance by measuring drive response times. With careful parameterization, a RAFT can handle the real-world challenges of network and drive operation latency for realistic file sizes and drive-cluster sizes.

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.

While a RAFT is attractive as a standalone tool or as a component in new storage-system designs, we have not explored the details of its integration into existing storage architectures. Cloud systems include layers of middleware and hardware controllers that pose their own delicate issues. Additionally, our RAFT system works most conveniently for a storage system dedicated to a single cloud tenant: The need for a brief system interruption to execute a RAFT could be inconvenient for multi-tenant storage devices. We leave the challenges of such scenarios to future work.

[8]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 Mike Luby and Amin Shokrollahi for their help in procuring a Raptor Code software package for our experiments. Thanks as well 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.

References

[1] 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.

[2] 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.

[3] 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, New York, NY, USA, 2009. ACM.

[4] S. Brands and D. Chaum. Distance-bounding protocols (extended abstract). In T. Helleseth, editor, *Proc. EUROCRYPT '93*, pages 344–359. Springer, 1993. LNCS vol. 765.

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

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

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

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

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

[17] 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.

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

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

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

[21] 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.

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

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

[24] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

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

APPENDIX

We consider adversaries who, due to resource constraints, do not pre-fetch blocks before the protocol starts. However, we may assume that during the duration of the whole protocol an adversary may fetch additional file blocks into fast access memory such that blocks corresponding to indices in a query Q that have already been additionally fetched can be read in negligible time. The server could also fetch file blocks if we allow for longer response times to account for variability in network latency, as discussed in Section V-A.

Let Z denote the indices of the fetched blocks. Notice that blocks in Z are fetched from idle drives in addition to the blocks that are needed to answer queries. In our lock-step protocol from Section VI, queries cannot be predicted ahead of time. For this reason we may assume that Z is chosen independent from each of the protocol's challenge queries.

Let z be defined as the number of pre-fetched blocks in Z that are at most stored in any subset of t drives. Let U be the set of indices of encoded file blocks that are stored in a fixed set of $d-t$ out of d server drives, and let U^* be the set of indices of encoded file blocks that are stored in the remaining t drives (notice that $U \cap U^*$ may not be empty if blocks are duplicated across drives). Then, by the definition of z, $|Z \setminus U| \leq |Z \cap U^*| \leq z$.

Lemma 4 (Fetching): Let $\hat{\alpha}$ be defined by the relation $n = (1+\hat{\alpha})(m+z)$. If

$$\epsilon_{\mathcal{S}} \leq B(c,t,\hat{\alpha}) = \frac{\hat{\alpha}(c-t)-t}{(1+\hat{\alpha})(c-t)},$$

then $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

We conclude that additional fetching can be compensated for by increasing the redundancy of the erasure code to incorporate $\hat{\alpha}$.

Proof: As in the proof of Lemma 3, we denote by X a fixed set of t drives. Let Y be the $d-t$ drives of $\mathcal{S}$ not in set X and let U be the set of indices of file blocks stored on drives in Y. Let $\beta = |U|/n$ be the fraction of file blocks that are stored on drives in Y.

Let $\gamma = |U \cup Z|/n$ be the fraction of file blocks in $U \cup Z$. With a similar argument as in Lemma 3, we can lower bound $\gamma \geq (1-\epsilon_{\mathcal{S}})(c-t)/c$.

Since $|U| = |U \cup Z| - |Z \setminus U| \geq |U \cup Z| - z$, fraction $\beta \geq \gamma - (z/n)$. Together with the lower bound on γ this shows that: if $(1-\epsilon_{\mathcal{S}})(c-t)/c \geq (m+z)/n$ (which is equivalent to the bound on $\epsilon_{\mathcal{S}}$ as stated in the lemma), then $\beta \geq m/n$, that is, file F can be reconstructed with the aid of the erasure code from the blocks stored on drives in Y. Since the bound on $\epsilon_{\mathcal{S}}$ does not depend on the allocation of the $d-t$ drives, we conclude that $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$. ■

The amount of additional fetching is restricted by the expected amount of (fast sequential) fetching from each of the server's drives during the duration of the whole protocol. In the case of an adversary who does not duplicate blocks

across drives, only the drive that stores a given block can fetch that block. Therefore, z is bounded by the number of blocks that can be read by t drives during the duration of the protocol. If both the number of reads during the amount of extra time acquired by pretending a larger network latency as well as the number of drives and queries are small compared to the file size, then $\hat{\alpha} \approx \alpha$ and the effect of additional fetching can be neglected.

In the cheap-and-lazy model we assume that the server stores only unaltered blocks of the encoded file on disk and does not computationally process blocks read from disk. In this appendix we consider servers who may computationally process blocks read from disk in order to answer queries. We will show how properly constrained rational adversaries can be reduced to cheap-and-lazy adversaries in a slight variation of the simple scheme.

The new scheme restricts the query space as follows. During encoding we select n as a multiple of c, and we construct sets of encoded file block indices

$$C_i = \{(i-1)n/c+1, (i-1)n/c+2, \ldots, in/c\},\ 1 \leq i \leq c.$$

We restrict query space $\mathcal{Q}$ to the n/c queries

$$\{i, n/c+i, \ldots, (c-1)n/c+i\},\ 1 \leq i \leq n/c.$$

We notice that queries in $\mathcal{Q}$ are disjoint and cover each index exactly once. Challenge queries are uniformly distributed over $\mathcal{Q}$. It is easy to show that the modified scheme satisfies the same completeness and advantage as the basic RAFT scheme.

Let $\{Q_1, \ldots, Q_k\} \subseteq \mathcal{Q}$ be the set of queries for which the adversary is able to compute correct answers by performing at most single block reads across all of its drives. Let D_i denote the set of indices of blocks that the adversary will read from its drives in order to correctly answer query Q_i by only using a single block read for each drive. An index in D_i specifies a drive and the location on its disk where the block corresponding to the index is stored. We notice that the stored blocks can be the result of complex manipulations of encoded file blocks. Also notice that answers are computed by using at most one block from each drive, which implies $|D_i| \leq d$, where d is the number of adversarial drives.

We want to construct a mapping from indices of encoded file blocks to positions on disks such that a cheap-and-lazy adversary responds to queries in the same way as the rational adversary would do. We start by defining

$$D(x) = D_i,\ \text{for } x \in Q_i.$$

Since all Q_i are disjoint, mapping D is well-defined. Let $\Delta = \cup_{i=1}^{k} Q_i$ be the domain of mapping D. Since $|D_i| \leq d$,

$$|D(x)| \leq d,\ \text{for } x \in \Delta. \tag{1}$$

The next lemma, which we prove in Appendix B, shows the existence of a mapping from file block indices to positions on disk.

Lemma 5 (Mapping): Let $D \in \Delta \rightarrow 2^{\Delta}$, i.e., D maps elements $x \in \Delta$ to subsets $D(x) \subseteq \Delta$. We may extend mapping D to $D \in 2^{\Delta} \rightarrow 2^{\Delta}$ by defining $D(X) = \cup_{x \in X} D(x)$ for subsets $X \subseteq \Delta$.

There exist sets $\Theta \subseteq \Delta$ and $\Theta^* = \Delta \setminus \Theta$ such that there exists an injective mapping $\theta \in \Theta \rightarrow D(\Delta)$ having the property

$$\theta(x) \in D(x),\ \text{for all } x \in \Theta,$$

and such that there exists a subset $K \subseteq \Theta$ having the property $D(\Theta^*) \subseteq D(K) = \theta(K)$ (here $\theta(K) = \{\theta(x) : x \in K\}$).

We define a cheap-and-lazy adversary by using mapping θ. Encoded file blocks with indices in Θ^* are not stored on its disks at all, and encoded file blocks with indices x in Θ are stored at the positions indicated by $\theta(x)$. Whenever a query $Q_i \subseteq \Theta$ is processed, the cheap-and-lazy adversary reads out and returns the encoded file blocks at positions indicated by $\theta(x)$ for $x \in Q_i$. According to the lemma, $\theta(x) \in D(x) = D_i$. This shows that the cheap-and-lazy adversary is able to answer queries, that are subsets of Θ, by accessing its disks in the same way as the rational adversary would access its disks.

Let $1-\epsilon$ be the "single-reads" probability of a query in $\mathcal{Q}$ which the adversarial server is able to answer by performing at most single block reads across all of its drives. If such a query is a subset of Θ, then the cheap-and-lazy adversary is also able to provide an answer by using single block reads across its drives. At most $|\Theta^*|$ out of the n/c possible queries in $\mathcal{Q}$ are not a subset of Θ and cannot be answered at all. We obtain that

$$1 - \epsilon - |\Theta^*|c/n$$

is at least the single-reads probability of a query in $\mathcal{Q}$ which the cheap-and-lazy server is able to answer by performing at most single block reads across all of its drives.

We want to estimate $|\Theta^*|$. Let G be the set of encoded file blocks g_x. By using Lemma 5, we define the set of blocks

$$G' = \{g_x : x \notin \Delta \text{ or } x \in \Theta \backslash K\} \cup \{\text{ blocks stored in } D(K)\}.$$

Let $x \in K \cup \Theta^*$. Since $D(\Theta^*) \subseteq D(K)$, the stored blocks indexed by $D(K)$ contain the blocks indexed by $D(x)$ which are used to compute the encoded file block indexed by x. From (1) we infer that each encoded file block in G can be computed by accessing G' at most d times. Since θ is injective, $|D(K)| = |\theta(K)| = |K|$, hence, $|G'| = n - |\Delta| + |\Theta \setminus K| + |K| = n - |\Theta^*|$. Therefore, if the modified scheme uses a $(n-u-1, d)$-incompressible IEC, then $n - |\Theta^*| \geq n - u$, and the single-reads probability of the cheap-and-lazy adversary is at least $1 - \epsilon - uc/n$.

Lemma 6 (Reduction): Let $1 - \epsilon_{\mathcal{S}}$ be the single-reads probability of a server $\mathcal{S}$ with d drives for the simple scheme using the slightly modified query space and using a

$(n-u-1,d)$-incompressible IEC. Then, there exists a cheap-and-lazy adversary $\mathcal{S}'$, who is restricted by the resources of $\mathcal{S}$ and has single-reads probability $1-\epsilon_{\mathcal{S}'} \geq 1-\epsilon_{\mathcal{S}}-uc/n$.

The reduction leads us to the main theorem on rational adversaries:

A. Main Result

Before stating the main result on the advantage of rational adversaries, we need the following definition and lemma regarding resource constraints:

Definition 3: Resource constraint (ρ, R) is μ-differentiable if $|\epsilon_0 - \epsilon_1| > \mu$ for $1-\epsilon_b$, $b \in \{0,1\}$, defined as the maximal single-reads probability that can possibly be achieved by a (ρ,R)-constrained server $\mathcal{S}$ with $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = b$.

Lemma 7: A (ρ,R)-constrained rational server has single-reads probability $1-\min\{\epsilon_0,\epsilon_1\}$.

Proof: Let $\mathcal{S}$ be a (ρ,R)-constrained rational adversary with single-reads probability $1-\epsilon_{\mathcal{S}}$. Given the constraint (ρ,R), $\mathcal{S}$ maximizes

$$\begin{aligned}\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}] &= \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\mathsf{NotFT}_{\mathcal{S}}]\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] + \\ &\quad \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\neg\mathsf{NotFT}_{\mathcal{S}}]\mathsf{Pr}[\neg\mathsf{NotFT}_{\mathcal{S}}]. \quad (2)\end{aligned}$$

If $\mathsf{Pr}[\neg\mathsf{NotFT}_{\mathcal{S}}] \neq 0$, then $\mathcal{S}$ induces a new server $\mathcal{S}'$, who uses $\mathcal{S}$ in $\mathsf{Exp}^{\mathcal{RAFT}(t)}_{\mathcal{S}'}(m,\ell,t)$ by asking $\mathcal{S}$ to generate a file placement $(d,\{\mathcal{D}_j\}_{j=1}^d)$ until the event $\neg\mathsf{NotFT}_{\mathcal{S}}$ is generated. This yields $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}'}] = \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\neg\mathsf{NotFT}_{\mathcal{S}}]$. Since $\mathcal{S}$ maximizes $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$, $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\neg\mathsf{NotFT}_{\mathcal{S}}] = \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}'}] \leq \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$. If $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] \neq 0$, then a similar argument yields $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\mathsf{NotFT}_{\mathcal{S}}] \leq \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$. Both arguments combined with (2) shows that if $\mathsf{Pr}[\neg\mathsf{NotFT}_{\mathcal{S}}] \neq 0$, then $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}'}] = \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}|\neg\mathsf{NotFT}_{\mathcal{S}}] = \mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$ together with $\mathsf{Pr}[\neg\mathsf{NotFT}_{\mathcal{S}'}] = 1$, that is, $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}'}] = 0$. Since $\mathcal{S}$ is rational, it first maximizes $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$ and next minimizes $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}]$. So, if $\mathsf{Pr}[\neg\mathsf{NotFT}_{\mathcal{S}}]$ is not equal to 0, then the existence of $\mathcal{S}'$ proves $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$. This shows that $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] \in \{0,1\}$. Now the lemma follows from the observation that in our scheme maximizing $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$ is equivalent to maximizing the single-reads probability $1-\epsilon_{\mathcal{S}}$. ∎

Differentiability measures into what extend requiring t fault tolerance versus not requiring t fault tolerance affects the maximal possible single-reads probability. If we assume that resource constraints are produced by the environment and not by clever design of a rational adversary, then we conjecture that it is likely that the amount of differentiability is not restricted to being very small. Our main result assumes that the differentiability can be at least uc/n, which is more likely for small u.

Theorem 3 (Main): In the simple scheme using the slightly modified query space and using a $(n-u-1,d)$-incompressible IEC, if a uc/n-differentiable (ρ,R)-constrained rational adversary $\mathcal{S}$ with d drives has single-reads probability $1-\epsilon_{\mathcal{S}}$ such that

$$\epsilon_{\mathcal{S}} \leq \frac{\alpha(c-t)-t}{(1+\alpha)(c-t)} - \frac{uc}{n},$$

then $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

Proof: Lemma 6 states that there exists a cheap-and-lazy adversary $\mathcal{S}'$ with single-reads probability $1-\epsilon_{\mathcal{S}'} \geq 1-\epsilon_{\mathcal{S}}-uc/n$. By the bound on $\epsilon_{\mathcal{S}}$ stated in the theorem, application of Lemma 3 proves $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}'}] = 0$. Therefore $\epsilon_0 \leq \epsilon_{\mathcal{S}} + uc/n$ (by the definition of ϵ_b for $b \in \{0,1\}$). Lemma 7 states $\epsilon_{\mathcal{S}} = \min\{\epsilon_0,\epsilon_1\}$. We conclude $|\epsilon_0 - \min\{\epsilon_0,\epsilon_1\}| = \epsilon_0 - \epsilon_{\mathcal{S}} \leq uc/n$. Since the resource constraint is uc/n-differentiable, $|\epsilon_0-\epsilon_1| > uc/n$. So, $\epsilon_{\mathcal{S}} = \epsilon_0$ from which we obtain $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$. ∎

Corollary 1: If in view of a rational adversary $\mathcal{S}$ with single-reads probability $1-\epsilon_{\mathcal{S}}$ encoded file blocks are random and independent of one another, then $\epsilon_{\mathcal{S}} \leq B(c,t,\alpha)$ implies $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

Proof: To such an adversary the used error correcting code cannot be compressed at all. So, we may apply the theorem for $u=0$. ∎

Theorem 3 can also be cast in a framework that allows more relaxed rational adversaries.

Definition 4: A (ρ,R)-constrained server $\mathcal{S}$ is μ-rational if: 1) If $\epsilon_0 > \epsilon_1+\mu$, then $\mathcal{S}$ achieves single-reads probability $1-\epsilon_1$. 2) If $\epsilon_0 \leq \epsilon_1+\mu$, then $\mathcal{S}$ achieves single-reads probability $1-\epsilon_0$ with $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

Lemma 7 and its proof show that being 0-rational coincides with being rational as defined in Definition 1. The relaxation towards μ-rational assumes that first $\mathcal{S}$ wants to maximize $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$. Second, if $\mathcal{S}$ is able to guarantee $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$ by achieving a slightly smaller $\mathsf{Pr}[\mathsf{Acc}_{\mathcal{S}}]$ (below the possible maximum), then $\mathcal{S}$ will do so. The adversary is economically motivated to pass the protocol a little less often in order to guard itself against drive failures (which increases the probability of being able to retrieve and return file F when needed).

Theorem 4: In the simple scheme using the slightly modified query space and using a $(n-u-1,d)$-incompressible IEC, if a (ρ,R)-constrained uc/n-rational adversary $\mathcal{S}$ with d drives has single-reads probability $1-\epsilon_{\mathcal{S}}$ such that

$$\epsilon_{\mathcal{S}} \leq \frac{\alpha(c-t)-t}{(1+\alpha)(c-t)} - \frac{uc}{n},$$

then $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

Proof: See the proof of Theorem 3, $\epsilon_0 \leq \epsilon_{\mathcal{S}} + uc/n \leq \epsilon_1 + uc/n$. By the definition of being uc/n-rational, $\mathsf{Pr}[\mathsf{NotFT}_{\mathcal{S}}] = 0$. ∎

For completeness, we notice that for $d=c$, any two sets D_i and D_j, $i \neq j$, are disjoint: Notice that D_i has sufficient information to reconstruct Q_i, so, $c = |Q_i| \leq |D_i| \leq c$. By using a similar argument, $|D_j| = c$. The union $D_i \cup D_j$ has sufficient information to reconstruct $Q_i \cup Q_j$. Since Q_i and

Q_j are disjoint, $2c = |Q_i| + |Q_j| = |Q_i \cup Q_j| \leq |D_i \cup D_j|$. Together with $|D_i| = |D_j| = c$ this proves $D_i \cap D_j = \emptyset$. Now we can easily construct an injective mapping θ which maps indices in Q_i to indices in D_i. We obtain:

Lemma 8: A server $\mathcal{S}$ with $d = c$ drives and double-read probability $\epsilon_{\mathcal{S}} \leq B(c, t, \alpha)$ has $\Pr[\mathsf{NotFT}_{\mathcal{S}}] = 0$.

For $d < c$, queries can only be answered by reading some disk at least twice:

Lemma 9: A server $\mathcal{S}$ with $d < c$ drives has double-read probability $\epsilon_{\mathcal{S}} = 1$.

We notice that the results of this appendix are easily generalized to adversaries who fetch additional blocks as described in Lemma 4.

B. Construction of θ

Let $W \subseteq \Delta$. We will answer the question for which W there exists an injective mapping $\theta \in W \to D(\Delta)$ such that

$$\theta(x) \in D(x), \text{ for all } x \in W. \tag{3}$$

If $W = \emptyset$, then (3) is satisfied and an injective mapping θ exists.

Suppose that there exists an injective mapping θ on W, which satisfies (3). Let $x \in \Delta \setminus W$. We will first prove that if x has a certain to be defined property, then θ can be transformed into an injective mapping θ' on $W \cup \{x\}$, which satisfies (3) with W replaced by $W \cup \{x\}$. The proof is not specific to the choice of $W \subseteq \Delta$. For this reason, we may conclude that there exists a mapping θ as stated in the lemma where W is extended to some set $W = \Theta$ such that none of the elements $x \in \Delta \setminus W = \Theta^*$ satisfies the yet to be defined property. The second part of the proof uses this fact to construct a mapping $k \in \Theta^* \to 2^{\Theta}$ with which we will construct a subset $K \subseteq \Theta$ for which $D(\Theta^*) \subseteq D(K) = \theta(K)$.

In order to construct an injective mapping θ' on $W \cup \{x\}$, we start by introducing some notation. First, we define $\theta(X) = \{\theta(x) : x \in X\}$ for subsets $X \subseteq W$. Second, we define a directed graph with vertices $\Delta \cup \{\perp\}$ and directed edges

$$y \to y' \text{ iff } y' \in W \text{ and } \theta(y') \in D(y),$$

and

$$y \to \perp \text{ iff } D(y) \setminus \theta(W) \neq \emptyset.$$

Suppose that there exists a path

$$x = y_1 \to y_2 \to \ldots \to y_H \to \perp. \tag{4}$$

We remind the reader that $x = y_1 \in \Delta \setminus W$. From the definition of our directed graph, we infer that

$$y_{h+1} \in W \text{ and } \theta(y_{h+1}) \in D(y_h), \text{ for } 1 \leq h \leq H-1,$$

and there exists

$$y \in D(y_H) \setminus \theta(W).$$

We define θ' as θ, except for the re-assignments:

$$\begin{aligned}
x = y_1 \in \Delta \setminus W &\quad\to\quad \theta'(y_1) = \theta(y_2) \in D(y_1), \\
y_2 \in W &\quad\to\quad \theta'(y_2) = \theta(y_3) \in D(y_2), \\
&\quad\ldots, \\
y_{H-1} \in W &\quad\to\quad \theta'(y_{H-1}) = \theta(y_H) \in D(y_{H-1}), \\
y_H \in W &\quad\to\quad \theta'(y_H) = y \in D(y_H) \setminus \theta(W).
\end{aligned}$$

Since θ is injective on W, the re-assignment defines a mapping θ' which is injective on $W \cup \{x\}$. We also notice that the definition of θ' satisfies (3) for W replaced by $W \cup \{x\}$.

As long as there exists an element x in $\Delta \setminus W$, we extend W to $W \cup \{x\}$ and we apply the re-assignment procedure to update θ. Let $\Theta \subseteq \Delta$ be a maximal subset of Δ to which W can be extended. This results into an injective mapping θ which satisfies (3) for $W = \Theta$.

Now, we will construct a mapping $k \in \Theta^* \to 2^{\Theta}$ having the property

$$D(x) \subseteq D(k(x)) = \theta(k(x)), \text{ for all } x \in \Theta^*. \tag{5}$$

We will use k to construct set K.

Let $x \in \Delta \setminus \Theta = \Theta^*$. We consider the directed graph defined for $W = \Theta$. Let $X \subseteq \Delta$ be the set of vertices that can be reached by x (in particular, $x \in X$). We define

$$k(x) = \Theta \cap X. \tag{6}$$

We will now prove (5).

Since Θ cannot be extended any further, we know that there does not exist a path as in (4). So, $\perp \notin X$ and no $y \in X$ can reach $\perp$. In particular, there does not exist an edge $y \to \perp$, which by the definition of the directed graph with $W = \Theta$ is equivalent to

$$D(y) \subseteq \theta(W) = \theta(\Theta), \text{ for } y \in X. \tag{7}$$

Let $y \in X$. Suppose that $D(y) \setminus \theta(\Theta \cap X)$ has an element z. Since θ is injective on Θ, we infer from (7) that $z \in D(y) \cap \theta(\Theta \setminus X)$. That is, there exists a $y' \in \Theta \setminus X$ such that $z = \theta(y') \in D(y)$, or equivalently, there exists an edge $y \to y'$ in the directed graph defined for $W = \Theta$. Since y can be reached by x, y' can be reached by x, that is, $y' \in X$ by the definition of set X. This contradicts $y' \in \Theta \setminus X$, and we conclude that the assumption $D(y) \setminus \theta(\Theta \cap X) \neq \emptyset$ is false. This proves

$$D(y) \subseteq \theta(\Theta \cap X) = \theta(k(x)), \text{ for all } y \in X. \tag{8}$$

In particular, $D(x) \subseteq \theta(\Theta \cap X) = \theta(k(x))$.

Since $D(k(x))$ is the union of all $D(y)$ for $y \in k(x) = \Theta \cap X$, (8) proves $D(k(x)) \subseteq \theta(k(x))$. Since $k(x) \subseteq \Theta$, (3) for $W = \Theta$ proves $\theta(k(x)) \subseteq D(k(x))$. We conclude (5), that is, $D(x) \subseteq D(k(x)) = \theta(k(x))$ for $x \in \Delta \setminus \Theta = \Theta^*$.

We use mapping k to define $K = \cup_{x \in \Theta^*} k(x) \subseteq \Theta$. By using (5), $D(\Theta^*) = \cup_{x \in \Theta^*} D(x) \subseteq \cup_{x \in \Theta^*} D(k(x)) =$

$D(K)$, and $\cup_{x\in\Theta^*} D(k(x)) = \cup_{x\in\Theta^*} \theta(k(x)) = \theta(K)$. Chaining the equations completes the proof of Lemma 5.

The complement of Definition 2 on incompressibility is:

Definition 5: 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)-compressible code (for $u < n$, $r < n$) if there exists an A such that $\mathsf{Adv}_A^{IEC}(m,n, \ell;u,r)$ is non-negligible.

Lemma 10: A $[n,m,n-m+1]$ RS code is $(n-u, r = \lceil (u+1)(n-u)/n \rceil)$-compressible for $u \le n-m$.

Proof: Notice that a $[n,m,n-m+1]$ RS code is a subcode of a $[n,n-u,u+1]$ RS code $\mathcal{C}$. Let G be the generator matrix of $\mathcal{C}$ that is used to encode message vectors v. If we show that $\mathcal{C}$ can also be generated by a generator matrix $G' = AG$ with at most r non-zero entries in each column for some invertible matrix A, then any code word symbol in vG can be computed by accessing at most r entries in vA. This would prove that $\mathcal{C}$ and its subcodes are $(n-u,r)$-compressible.

Any subset S of $n-u$ code word symbols in $\mathcal{C}$ are information symbols. Hence, for $i \in S$, there exists a code word that has zeroes in each of the $n-u-1$ positions indicated by $S \setminus \{i\}$ and has a non-zero code word symbol in position i. Such a code word has Hamming weight at most $n-(n-u-1) = u+1$ and Hamming weight at least the minimum distance $u+1$. By selecting appropriate subsets S, we can construct a generator matrix G' of $n-u$ such code words each having exactly $u+1$ non-zero entries and such that the collection of all $n-u$ code words distributes the non-zero entries evenly over the columns. That is, each column in G' has at most $r = \lceil (u+1)(n-u)/n \rceil$ non-zero positions. ■