PORs: Proofs of Retrievability for Large Files

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

Burton S. Kaliski Jr.
EMC Corporation
Hopkinton, MA, USA
kaliski_burt@emc.com

ABSTRACT

In this paper, we define and explore the notion of a *proof of retrievability* (POR). A POR enables an archive or back-up service (prover) to demonstrate to a user (verifier) that it has "possession" of a file F, that is, that the archive retains data sufficient for the user to retrieve F in its entirety.

A POR may be viewed as a kind of cryptographic proof of knowledge (POK), but one specially designed to handle a *large* file (or bitstring) F. We explore POR protocols here in which the communication costs, number of memory accesses for the prover, and storage requirements of the user (verifier) are small parameters essentially independent of the length of F. In addition, in a POR, unlike a POK, neither the prover nor the verifier need actually have knowledge of F. PORs give rise to a new and unusual security definition.

We view PORs as an important tool for the management of semi-trusted online archives. Existing cryptographic tools help users ensure the privacy and integrity of their files once they are retrieved. It is also natural, however, for users to want to verify that archives do not delete or modify files while they are stored. The goal of a POR is to accomplish these checks *without users having to download the files themselves*. A POR can also provide quality-of-service guarantees, i.e., show that a file is retrievable within a certain time bound.

1. INTRODUCTION

Several trends are opening up computing systems to new forms of outsourcing, that is, delegation of computing services to outside entities. Improving network bandwidth and reliability are reducing user reliance on local resources. Energy and labor costs as well as computing-system complexity are militating toward the centralized administration of hardware. Increasingly, users employ software and data that reside thousands of miles away on machines that they themselves do not own. Grid computing, the harnessing of disparate machines into a unified computing platform, has played a role in scientific computing for some years. Similarly, remotely administered application software—loosely a throwback to terminal/mainframe computing architectures—is now a pillar in the internet-technology strategies of major companies like Google and Microsoft.

Storage is no exception to the outsourcing trend. Online data-backup services abound for consumers and enterprises alike. Amazon Simple Storage Service (S3) [1], for example, offers an abstracted online-storage interface, allowing programmers to access data objects through web-service calls, with fees metered in gigabyte-months and data-transfer amounts. Researchers have investigated alternative service models, such as peer-to-peer data archiving [10].

As users and enterprises come to rely on diverse sets of data repositories, with variability in service guarantees and underlying hardware integrity, they will require new forms of assurance of the integrity and accessibility of their data. Simple replication offers one avenue to higher-assurance data archiving, but at often unnecessarily and unsustainably high expense. (Indeed, a recent IDC report suggests that data-generation is outpacing storage availability [12].) Protocols like Rabin's data-dispersion scheme [29] are more efficient: They share data across multiple repositories with minimum redundancy, and ensure the availability of the data given the integrity of a quorum (k-out-of-n) of repositories. Such protocols, however, do not provide assurances about the state of *individual repositories*—a shortcoming that limits the assurance the protocols can provide to relying parties.

In this paper, we develop a new cryptographic building block known as a *proof of retrievability* (POR). A POR enables a user (verifier) to determine that a prover (archive) "possesses" a file or data object F. More precisely, a successfully executed POR assures a verifier that the prover presents a protocol interface through which the verifier can retrieve F in its entirety. Of course, a prover can refuse to release F even after successfully participating in a POR. A POR, however, provides the strongest possible assurance of file retrievability barring changes in prover behavior.

As we demonstrate in this paper, a POR can be efficient enough to provide regular checks of file retrievability. Consequently, as a general tool, a POR can complement and strengthen any of a variety of archiving architectures, including those that involve data dispersion.

1.1 A first approach

To illustrate the basic idea and operation of a POR, it is worth considering a straightforward design involving a keyed hash function $h_k(F)$. In this scheme, prior to archiving a file F, the verifier computes and stores a hash value $r = h_k(F)$ along with secret, random key k. To check that the prover possesses F, the verifier releases k and asks the prover to compute and return r. Provided that h is resistant to second-preimage attacks, this simple protocol provides a strong proof that the prover knows F. By storing multiple hash values over different keys, the verifier can initiate multiple, independent checks.

This keyed-hash approach, however, has an important drawback: High resource costs. The keyed-hash protocol requires that the verifier store a number of hash values linear in the number of checks it is to perform. This characteristic conflicts with the aim of enabling the verifier to offload its storage burden. More importantly, each protocol invocation requires that the prover process the *entire* file F. For

large F, even a computationally lightweight operation like hashing can be highly burdensome.

1.2 Our approach

We introduce a POR protocol in which the verifier stores only a single cryptographic key—irrespective of the size and number of the files whose retrievability it seeks to verify—as well as a small amount of dynamic state (some tens of bits) for each file. (One simple variant of our protocol allows for the storage of no dynamic state, but yields weaker security.) More strikingly, and somewhat counterintuitively, our scheme requires that the prover access only a small portion of a (large) file F in the course of a POR. In fact, the portion of F "touched" by the prover is essentially independent of the length of F and would, in a typical parameterization, include just hundreds or thousands of data blocks.

Briefly, our POR protocol encrypts F and randomly embeds a set of randomly-valued check blocks called *sentinels*. The use of encryption here renders the sentinels indistinguishable from other file blocks. The verifier challenges the prover by specifying the positions of a collection of sentinels and asking the prover to return the associated sentinel values. If the prover has modified or deleted a *substantial* portion of F, then it will also have suppressed a number of sentinels. It is therefore unlikely to respond correctly to the verifier. To protect against corruption by the prover of a *small* portion of F, we also employ error-correcting codes. We let $\tilde{F}$ refer to the full, encoded file stored with the prover.

A drawback of our proposed POR scheme is the preprocessing / encoding of F required prior to storage with the prover. This step imposes some computational overhead—beyond that of simple encryption or hashing—as well as larger storage requirements on the prover. The sentinels may constitute a small fraction of the encoded $\tilde{F}$ (typically, say, 2%); the error-coding imposes the bulk of the storage overhead. For large files and practical protocol parameterizations, however, the associated expansion factor $|\tilde{F}|/|F|$ can be fairly modest, e.g., 15%.

To illustrate the intuition behind our POR protocol a little better, we give two brief example scenarios.

EXAMPLE 1. *Suppose that the prover, on receiving an encoded file $\tilde{F}$, corrupts three randomly selected bits, $\beta_1, \beta_2, \beta_3$. These bits are unlikely to reside in sentinels, which constitute a small fraction of $\tilde{F}$. Thus, the verifier will probably not detect the corruption through POR execution. Thanks to the error-correction present in $\tilde{F}$, however, the verifier can recover the original file F completely intact.*

Suppose conversely that the prover corrupts many blocks in $\tilde{F}$, e.g., 20% of the file. In this case (absent very heavy error-coding), the verifier is unlikely to be able to recover the original file F. On the other hand, every sentinel that the verifier requests in a POR will detect the corruption with probability about 1/5. By requesting hundreds of sentinels, the verifier can detect the corruption with overwhelming probability.

1.3 Related work

File-integrity assurance is of course one of the fundamental goals of cryptography. Primitives such as digital signatures and message-authentication codes (MACs) allow an entity in possession of a file F to verify that it has not been subjected to tampering.

A more challenging problem is to enable verification of the integrity of F without explicit knowledge of the full file. Clarke et al. [9], for instance, consider the problem of a trusted entity with a small amount of state, e.g., a trusted computing module, verifying the integrity of arbitrary blocks of untrusted, external, dynamically-changing memory. Their constructions employ a Merkle hash-tree over the contents of this memory, an approach that has seen very fruitful application elsewhere in the literature.

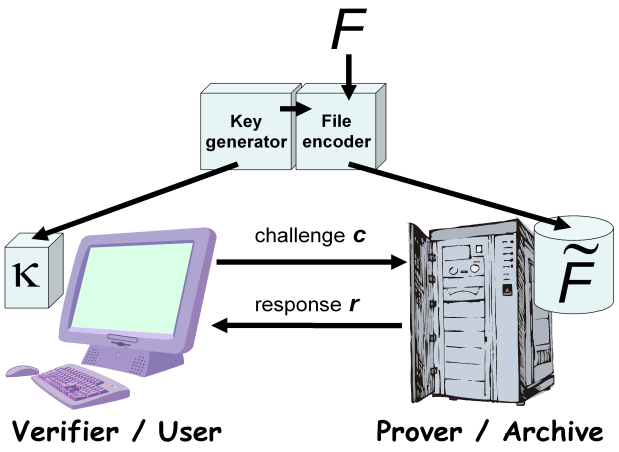

Figure 1: Schematic of a POR system. An encoding algorithm transforms a raw file F into an encoded file $\tilde{F}$ to be stored with the prover / archive. A key generation algorithm produces a key κ stored by the verifier / user and used in encoding. (The key κ is independent of F in some PORs, as in our main scheme.) The verifier performs a challenge-response protocol with the prover to check that the verifier can retrieve F.

In networked storage environments, cryptographic file systems (CFSs) are the most common tool for system-level integrity assurance (see, e.g., [20] for a good, recent survey). In a CFS, one entity, referred to as a *security provider*, manages the encryption and/or integrity-protection of files in untrusted storage providers. The security provider may be either co-located with a physical storage device or architected as a virtual file system.

Cryptographic integrity assurance allows an entity to detect unauthorized modifications to portions of files upon their retrieval. Such integrity assurance in its basic form does not enable the detection of modification or deletion of files *prior* to their retrieval or on an ongoing basis. It is this higher degree of assurance that a POR aims to provide.

A POR permits detection of tampering or deletion of a remotely located file—or relegation of the file to storage with poor service quality. A POR does not by itself, however, protect against loss of file contents. File robustness requires some form of storage redundancy and, in the face of potential system failures, demands the distribution of a file across multiple systems. A substantial literature, e.g., [3, 26, 27], explores the problem of robust storage in a security model involving a collection of servers exhibiting Byzantine behavior. The goal is simulation of a trusted read/write memory register, as in the abstraction of Lamport [22]. In such distributed models, the robustness guarantees on the simulated memory register depend upon a quorum of honest servers.

While many storage systems operating in the Byzantine-

failure model rely on storage duplication, an important recent thread of research involves the use of information dispersal [29] and error-coding to reduce the degree of file redundancy required to achieve robustness guarantees, as in [8]. Similarly, we use error-correction in our main POR construction to bound the effects of faults in a storage archive in our constructions.

While a POR only aims at detection of file corruption or loss, and not prevention, it can work hand-in-hand with techniques for file robustness. For example, a user may choose to disperse a file across multiple service providers. By executing PORs with these providers, the user can detect faults or lapses in service quality. She can accordingly redistribute her file across providers to strengthen its robustness and availability. In peer-to-peer environments, where service quality may be unreliable, such dynamic reallocation of resources can be particularly important.

As we explain in detail in section 2, a POR is loosely speaking a kind of proof of knowledge (POK) [4] created by an archive / prover for consumption by a user / verifier on a file F. A proof of knowledge serves to demonstrate knowledge by the prover of some short secret y that satisfies a predicate specified by the verifier. Generally, as in an authentication protocol, the essential design property of a POK is to preserve the secrecy of y, i.e., not to reveal information about y to the verifier. The concept of zero-knowledge [15, 16] captures this requirement in a strict, formal sense. In a POR, the design challenge is different. The verifier has potentially already learned the value F whose knowledge the prover is demonstrating (as the verifier may have encoded the file to begin with). Since F is potentially quite large, the main challenge is to prove knowledge of F using computational and communication costs substantially smaller than $|F|$.

As such, PORs are akin to other unorthodox cryptographic proof systems in the literature, such as proofs of computational ability [34] and proofs of work (POWs) [19]. Memory-bound POWs [11] are similar to the use of PORs for quality-of-service verification in that both types of proof aim to characterize memory use in terms of the latency of the storage employed by the prover. Very close in spirit to a POR is a construction of Golle, Jarecki, and Mironov [17], who investigate "storage-enforcing commitment schemes." Their schemes enable a prover to demonstrate that it is making use of storage space at least $|F|$. The prover does not prove directly that it is storing file F, but proves that it is has committed sufficient resources to do so (and therefore, barring malice, has an economic incentive to store F).

The use of sentinels in our main scheme is similar is spirit to a number of other systems that rely on the embedding of secret check values in files, such as the "ringers" used in [18]. There the check values are easily verifiable computational tasks that provide evidence for the correct processing of accompanying tasks. PORs bear an important operational difference in that they involve "spot checks" or auditing, that is, the prover is challenged to reveal check values in isolation from the rest of the file. The distinguishing feature of the POR protocols we propose here is the way that they amplify the effectiveness of spot-checking for the special case of file-verification by combining cryptographic hiding of sentinels with error-correction.

The only published POR protocol of which we are aware is that of Filho and Barreto [14]. Making indirect use of a homomorphic RSA-based hash introduced by Shamir [30], their scheme is as follows. Let N be an RSA modulus. The verifier stores $k = F \bmod \phi(N)$ for file F (suitably represented as an integer). To challenge the prover to demonstrate retrievability of F, the verifier transmits a random element $g \in \mathbb{Z}_N$. The prover returns $s = g^F \bmod N$, and the verifier checks that $g^k \bmod N = s$. This protocol has the drawback of requiring the prover to exponentiate over the entire file F.[1] Additionally, unlike the basic hash function, the security of the Filho-Barreto POR would appear to rely on some sort of strong, non-standard hardness assumption; there is no clear security reduction to the RSA problem or any well-known variant.

Organization

In section 2, we introduce a formal definition of a POR, and explain how this definition differs from the standard cryptographic view of proofs of knowledge. We introduce our main POR scheme in section 3, briefly discuss its security, and describe several variants. We describe the adaptation and application of our POR scheme to the problem of secure archiving and quality-of-service checking in section 4. We conclude in section 5 with a brief discussion of future research directions. We prove our main theorem in appendix A.

2. DEFINITIONS

2.1 Standard proof-of-knowledge definitions and PORs

Bellare and Goldreich (BG) established a standard, widely referenced definition of proofs of knowledge in [4]. Their definition centers on a binary *relation* $R \subseteq \{0,1\}^* \times \{0,1\}^*$.

A language $L_R = \{x : \exists\, y \text{ s.t. } (x,y) \in R\}$ is defined as the set of values x that induce valid relations. The set $R(x) = \{y : (x,y) \in R\}$ defines the *witnesses* associated with a given x. Typically, relations of interest are polynomial, meaning that the bitlength $|y|$ of any witness is polynomial in $|x|$.

In the BG view, a proof of knowledge is a two-party protocol involving a prover P and a verifier V. Each player is a probabilistic, interactive function. The BG definition supposes that P and V share a common string x. A transcript includes the sequence of outputs of both players in a given interaction.

The BG definition relies upon an additional function, an *extractor* algorithm K that also takes x as input and has oracle access to P. Additionally, V has an associated *error function* $\kappa(x)$, essentially the probability that V accepts transcripts generated by a prover P that does not actually know (or use its knowledge of) a witness for x. For every prover P, let $p(x)$ be the probability that on input x, prover P induces a set of transcripts that V accepts.

Briefly, then, in the BG definition, a poly-time verifier V characterizes a proof of knowledge with respect to a relation R and with error κ if the following holds: There exists a constant c such that for every prover P, the extractor K outputs a witness $y \in R(x)$ in expected time bounded by $|x|^c/(p(x) - \kappa(x))$. (The BG definition also has a non-triviality requirement: There must exist a legitimate prover

[1] In a brief research abstract, Burns [6] announces an RSA-based POR-like protocol that does not require processing of a full file, but details appear to be as yet unpublished.

P, i.e., a prover that causes V to accept with probability 1 for any $x \in L_R$.)

Intuitively, the BG definition states that if prover P can convince verifier V that $x \in L_R$, then P "knows" a witness y. The stronger P's ability to convince a verifier, the more efficiently a witness y can be extracted from P.

While very broad, the BG definition does not naturally capture the properties of POR protocols, which have several distinctive characteristics:

1. **No common string x:** In a POR, P and V may not share any common string x: P may merely have knowledge of some file F, while V possesses secret keys for verifying its ability to retrieve F from P and also for actually performing the retrieval.

2. **No natural relation R:** Since a POR aims to prove that the file F is subject to recovery from P, it would seem necessary to treat F as a witness, i.e., to let $y = F$, since F is precisely what we would like to extract. In this case, however, if we regard x as the input available to V, we find that there is no appropriate functional relation $R(x, y)$ over which to define a POR: In fact, x may be perfectly independent of F.

3. **Split verifier/extractor knowledge:** It is useful in our POR protocols to isolate the ability to verify from the ability to extract. Thus, K may take a secret input unknown to either P or V.

As we show, these peculiarities of PORs give rise to a security definition rather different than for ordinary POKs.

2.2 Defining a POR system

A *POR system* PORSYS comprises the six functions defined below. The function respond is the only one executed by the prover / archive P. All others are executed by the verifier / user V. For a given verifier invocation in a POR system, it is intended that the set of verifier-executed functions share and implicitly modify some persistent state α. In other words, α represents the state of a given invocation of V; we assume α is initially null. We let π denote the full collection of system parameters. The only parameter we explicitly require for our system and security definitions is a security parameter j. (In practice, as will be seen in our main scheme in section 3, it is convenient for π also to include parameters specifying the length, formatting, and encoding of files, as well as challenge/response sizes.) On any failure, e.g., an invalid input or processing failure, we assume that a function outputs the special symbol $\perp$.

$\mathsf{keygen}[\pi] \rightarrow \kappa$: The function keygen generates a secret key κ. (In a generalization of our protocol to a public-key setting, κ may be a public/private key pair. Additionally, for purposes of provability and privilege separation, we may choose to decompose κ into multiple keys.)

$\mathsf{encode}(F; \kappa, \alpha)[\pi] \rightarrow (\tilde{F}_\eta, \eta)$: The function encode generates a file handle η that is unique to a given verifier invocation. The function also transforms F into an (enlarged) file $\tilde{F}_\eta$ and outputs the pair $(\tilde{F}_\eta, \eta)$.

Where appropriate, for a given invocation of verifier V, we let F_η denote the (unique) file whose input to encode has yielded handle η. Where this value is not well defined, i.e., where no call by verifier V to encode has yielded handle η, we let $F_\eta \stackrel{def}{=} \perp$.

$\mathsf{extract}(\eta; \kappa, \alpha)[\pi] \rightarrow F$: The function extract is an interactive one that governs the extraction by verifier V of a file from an archive P. In particular, extract determines a sequence of challenges that V sends to P, and processes the resulting responses. If successful, the function recovers and outputs F_η.

$\mathsf{challenge}(\eta; \kappa, \alpha)[\pi] \rightarrow c$. The function challenge takes secret key κ and a handle and accompanying state as input, along with system parameters. The function challenge outputs a challenge value c for the file η.

$\mathsf{respond}(c, \eta, \tilde{F}) \rightarrow r$. The function respond is used by the archive P to generate a response to a challenge c. Note that in a POR system, a challenge c may originate either with challenge or extract.

$\mathsf{verify}((r, \eta); \kappa, \alpha) \rightarrow b \in \{0, 1\}$. The function verify determines whether r represents a valid response to challenge c. The challenge c does not constitute explicit input in our model; it is implied by η and the verifier state α. The function verify outputs a '1' bit if verification succeeds, and '0' otherwise.

A set of functions $\mathsf{PORSYS} = \{\mathsf{keygen}, \mathsf{encode}, \mathsf{extract}, \mathsf{challenge}, \mathsf{respond}, \mathsf{verify}\}$ represents a POR system.

2.3 POR security definition

We define the security of a POR protocol in terms of an experiment in which the adversary $\mathcal{A}$ plays the role of the archive P. Let us first give some preliminary explanation and intuition.

2.3.1 Definition overview

The adversary $\mathcal{A}$ consists of two parts, $\mathcal{A}$("setup") and $\mathcal{A}$("respond"). The function $\mathcal{A}$("setup") may interact arbitrarily with the user / verifier; it may create files and cause the verifier to encode and extract them; it may also obtain challenges from the verifier. The purpose of $\mathcal{A}$("setup") is to create an archive on a special file F_{η^*}. This archive is embodied as the second adversarial function $\mathcal{A}$("respond"). It is with $\mathcal{A}$("respond") that the verifier executes the POR and attempts to retrieve F_{η^*}.

In our model, an archive—whether honest or adversarial—performs only one function. It receives a challenge c specifying a position in an encoded file $\tilde{F}_\eta$ and returns a block of data. An honest archive returns the block in position c in $\tilde{F}_\eta$. An adversary may or may not return the correct file block. This challenge/response mechanism serves, of course, as the foundation for proving retrievability in a POR. Additionally, however, it is the interface by which the function extract recovers a file F_η. In the normal course of operation, extract can submit a sequence of challenges $1, 2, 3...$ to an archive, reconstruct $\tilde{F}_\eta$ from the corresponding responses, and then decode to obtain the original file F_η.

In our security definition, we regard $\mathcal{A}$("respond") as a *stateless* entity. On any given challenge c, $\mathcal{A}$("respond") returns the correct corresponding block from F_{η^*} with some probability; otherwise, it returns an incorrect block according to some fixed probability distribution. These probabili-

ties may be different from block to block, i.e., challenge to challenge, but because of our assumption that $\mathcal{A}$("respond") is stateless, the probabilities remain fixed for any given challenge value. Put another way, $\mathcal{A}$("respond") may be viewed as set of probability distributions over block values, with one such distribution for each possible challenge value c.

It may seem at first that the assumption of statelessness in $\mathcal{A}$("respond") is too strong. In practice, after all, since an extractor must send many more queries than a verifier, a stateful adversary can distinguish between the two. Thus, by assuming that $\mathcal{A}$("respond") is stateless, our definition discounts the (quite real) possibility of a malicious archive that responds correctly to a verifier, but fails to respond to an extractor. Such a stateful adversary responds correctly to challenges but still fails to release a file.

We believe, however, that our POR definition is among the strongest possible in a real-world operational environment and that it captures a range of useful, practical assurances. There is in fact no meaningful way to define a POR without assuming some form of restriction on adversarial behavior. As we have explained, unless the POR protocol is indistinguishable from extract, a Byzantine adversary can always fail when it detects an extraction attempt. Thus, the most appropriate security definition seems to be one that characterizes the ability of a verifier to extract a file F_η from a "snapshot," i.e., from the full state of $\mathcal{A}$("respond"). The verifier can then "rewind" the adversary as desired during execution to preserve the adversary's lack of state. In a real-world environment, this ability corresponds to access to a memory dump or backup tape for an adversarial server.

On the other hand, for some applications, our modeling of $\mathcal{A}$("respond") may actually seem too strong. For example, if the purpose of a POR is to ensure availability or quality-of-service in an archive, then there may be no reason to assume the possibility of adversarial corruption of a file. An adversary's only real economic incentive may be to minimize its storage requirements, i.e., to delete file blocks or relegate them to slow storage. Our POR security definition may be modified to meet this weaker requirement, and we in fact consider an "erasing" adversary later in the paper. Our concern here, however, is to create a foundational definition with broad real-world applicability.

Briefly, our security definition involves a game in which the adversary $\mathcal{A}$ seeks to "cheat" a verifier V. $\mathcal{A}$ tries to create an environment in which V believes that it will be able to retrieve a given file F_{η^*} with overwhelming probability, yet cannot. Thus the aim of $\mathcal{A}$("setup") is to induce a verifier state α and create state (δ, η^*) in $\mathcal{A}$("respond") such that: (1) V accepts responses from $\mathcal{A}$("respond") to challenges with high probability and (2) V fails with non-negligible probability to retrieve F_{η^*} from $\mathcal{A}$("respond") on invoking extract.

2.3.2 Definition details

We let $\mathcal{O}_{\mathsf{encode}}$, $\mathcal{O}_{\mathsf{extract}}$, $\mathcal{O}_{\mathsf{challenge}}$, and $\mathcal{O}_{\mathsf{verify}}$ represent oracles respectively for functions encode, extract, challenge, verify in a single invocation of verifier V. These oracles take κ and α as implicit inputs, i.e., inputs not furnished by (or revealed to) the adversary. Additionally, these oracles may, of course, modify state α, which is initially null. We denote by the symbol '$\cdot$' those input values that the adversary has the freedom to specify in its oracle calls.

We let '*' denote protocol values created by $\mathcal{A}$ in the course of our experiment. We let δ denote adversarial state—in particular, the state information passed from $\mathcal{A}$("setup") to $\mathcal{A}$("respond). As above, we let π denote the parameter set for a POR system PORSYS. Where appropriate for brevity, however, we drop π from our notation.

In our first experiment $\mathbf{Exp}^{setup}$, the adversary $\mathcal{A}$("setup") is permitted to interact arbitrarily with system oracles. At the end of the experiment $\mathcal{A}$("setup") specifies a file handle η^* and state δ as input to the adversarial function $\mathcal{A}$("respond") for the next experiment.

Experiment $\mathbf{Exp}^{setup}_{\mathcal{A},\mathsf{PORSYS}}[\pi]$
- $\kappa \leftarrow \mathsf{keygen}(j); \alpha \leftarrow \phi$; % generate key, initialize oracle state
- $(\delta, \eta^*) \leftarrow \mathcal{A}^{\mathcal{O}}$("setup"); % accessing $\mathcal{O}$, $\mathcal{A}$ creates archive
- output (α, δ, η^*) % output full oracle and archive states

In our next experiment, the adversarial archive $\mathcal{A}$("respond") responds to a challenge issued by the verifier. The adversary is deemed successful if it generates a response accepted by the verifier.

Experiment $\mathbf{Exp}^{chal}_{\mathcal{A},\mathsf{PORSYS}}(\alpha, \delta, \eta^*)[\pi]$
- $c^* \leftarrow \mathcal{O}_{\mathsf{challenge}}(\eta^*; \kappa, \alpha)$; % issue "challenge"
- $r^* \leftarrow \mathcal{A}(\delta, c^*)$("respond"); % adversary outputs response
- $\beta \leftarrow \mathcal{O}_{\mathsf{verify}}((r^*, \eta^*); \kappa, \alpha)$; % verify adversarial response
- output β % output '1' if response correct, otherwise '0'

We define $\mathbf{Succ}^{chal}_{\mathcal{A},\mathsf{PORSYS}}(\alpha, \delta, \eta^*)[\pi]$
$= \mathsf{pr}\big[\mathbf{Exp}^{chal}_{\mathcal{A},\mathsf{PORSYS}}(\alpha, \delta, \eta^*)[\pi] = 1\big]$, i.e., the probability that the adversarial archive succeeds in causing the verifier to accept.

Given these experiment specifications, we now specify our definition of security for a POR. Our definition aims to capture a key intuitive notion: That an adversary with high probability of success in $\mathbf{Exp}^{chal}$ must "possess" F_{η^*} in a form that may be retrieved by the verifier, i.e., by an entity with knowledge of κ and α. By analogy with security definitions for standard proofs of knowledge, we rely on the *extractor* function extract. One special feature of a POR systems is that extract *is not just a component of our security proof.* As we have already seen, it is *also* a normal component of the POR system. (For this reason, $\mathcal{A}$ actually has oracle access to extract in $\mathbf{Exp}^{setup}$.) A priori, $\mathcal{A}$("respond") cannot distinguish between challenges issued by challenge and those issued by extract.

In our security definition, the function extract is presumed to have oracle access to $\mathcal{A}(\delta, \cdot)$("respond"). In other words, it can execute the adversarial archive on arbitrary challenges. Since $\mathcal{A}$ is cast simply as a function, we can think of extract as having the ability to rewind $\mathcal{A}$. The idea is that the file F_{η^*} is retrievable from $\mathcal{A}$ if extract can recover it. In essence, the respond function of the archive / adversary is the interface by which extract recovers F.

We thus define

$$\mathbf{Succ}^{extract}_{\mathcal{A},\mathsf{PORSYS}}(\alpha,\delta,\eta^*)[\pi] = \mathsf{pr}\big[F = F_{\eta^*} \;\big|\; F \leftarrow \mathsf{extract}^{\mathcal{A}(\delta,\cdot)(\text{“respond”})}(\eta^*;\kappa,\alpha)[\pi]\big].$$

In other words, $\mathbf{Succ}^{extract}_{\mathcal{A},\mathsf{PORSYS}}$ is simply the probability that the extractor successfully recovers F_{η^*}.

Let a poly-time algorithm $\mathcal{A}$ be one whose running time is bounded by a polynomial in security parameter j. Our main security definition, then, is as follows.

DEFINITION 1. *A poly-time POR system* $\mathsf{PORSYS}[\pi]$ *is a* (ρ,λ)-valid proof of retrievability (POR) *if for any poly-time* $\mathcal{A}$ *and for some* ζ *negligible in security parameter* j*,*

$$\mathsf{pr}\left[\begin{array}{l}\mathbf{Succ}^{extract}_{\mathcal{A},\mathsf{PORSYS}}(\alpha,\delta,\eta^*) < 1-\zeta, \\ \mathbf{Succ}^{chal}_{\mathcal{A},\mathsf{PORSYS}}(\alpha,\delta,\eta^*) \geq \lambda\end{array} \;\middle|\; \begin{array}{l}(\alpha,\delta,\eta^*) \\ \leftarrow \mathbf{Exp}^{setup}_{\mathcal{A},\mathsf{PORSYS}}\end{array}\right] \leq \rho.$$

At an intuitive level, our definition establishes an upper bound ρ on the probability that $\mathcal{A}$(“setup”) generates a “bad” environment—i.e., a “bad” adversarial archive (δ,η^*) in $\mathcal{A}$(“respond”) and system state α in the verifier. We regard an environment as “bad” if the verifier accepts adversarial responses with probability at least λ, but extraction nonetheless fails with non-negligible probability ζ. In other words, in such an environment the verifier cannot extract F_{η^*} with overwhelming probability, but with probability at least λ is still convinced that it can do so. Note that we treat ζ asymptotically in our definition for simplicity: This treatment eliminates the need to consider ζ as a concrete parameter in our analyses.

2.3.3 Remarks

- Note that we do not give $\mathcal{A}$ oracle access in $\mathbf{Exp}^{chal}$. This is a simplifying assumption: We rule out the possibility of $\mathcal{A}$ learning additional information from the verifier through $\mathcal{O}_{verify}$ or periodic file retrievals. We believe this assumption to be reasonable in practice: Given strong cryptographic primitives, the information harvested by $\mathcal{A}$ through these channels should be negligible, although a broader model is worth exploration.
- We can also define a (ρ,λ)-valid proof of retrievability for an *erasing adversary*. Such an adversary is only permitted to reply to a challenge with either a correct response or a null one. In section 4, we consider an important practical setting for this variant.
- The purpose of encryption in our POR is to render sentinels indistinguishable from text blocks. A possible alternative is to embed sentinels as steganographic inclusions or “watermarks,” drawing them from a distribution indistinguishable from that of plaintext file blocks. This approach has the drawback of relying on file-data specific knowledge.

3. SENTINEL-BASED POR SCHEME

Our main POR scheme of interest is the sentinel-based one described in the introduction. At root, this POR protocol is conceptually simple. Before giving details, we outline the general protocol structure.

The verifier / user V encrypts the file F. It then embeds sentinels in random positions in F, sentinels being randomly constructed check values. Let $\tilde{F}$ denote the file F with its embedded sentinels.

To ensure that the archive has retained F, V specifies the positions of some sentinels in $\tilde{F}$ and asks the archive to return the corresponding sentinel values. Because F is encrypted and the sentinels are randomly valued, the archive cannot feasibly distinguish *a priori* between sentinels and portions of the original file F. Thus we achieve the following property: If the archive deletes or modifies a substantial, ϵ-fraction of $\tilde{F}$, it will with high probability also change roughly an ϵ-fraction of sentinels. Provided, then, that the verifier V requests and verifies enough sentinels, V can detect whether the archive has erased or altered a substantial fraction of $\tilde{F}$. (Individual sentinels are, however, only one-time verifiable.)

In practice, of course, a verifier / user wants to ensure against change to *any portion of the file* F. Even a single missing or flipped bit can represent a semantically significant corruption. Thus, detection of only ϵ-fraction modification is insufficient for our purposes in the basic protocol we have just described. With a simple trick, though, we can ensure that even if the archive *does* change an ϵ-fraction (for arbitrarily large ϵ), the verifier can still recover its file. Very simply, before planting sentinels in the file F, the user applies an *error-correcting code* that tolerates corruption (or erasure, if appropriate) of an ϵ-fraction of data blocks in $\tilde{F}$. The verifier also permutes the file to ensure that the symbols of the code are randomly dispersed, and therefore that their positions are unknown to the archive.

We emphasize one strongly counterintuitive aspect of our PORs scheme: The sentinels, which constitute the content of a POR proof, are in fact generated *independently of the bit-string whose retrievability they are proving.* By contrast, as explained above, in an ordinary proof of knowledge (POK), the content of a proof depends on the values that are the subject of the proof: Typically the prover derives the proof from a *witness*.

3.1 POR efficiency

Of course, application of an error-correcting (or erasure) code and insertion of sentinels enlarges $\tilde{F}$ beyond the original size of the file F. The expansion induced by our POR protocol, however, can be restricted to a modest percentage of the size of F. Importantly, the communication and computational costs of our protocol are low. As we mention below, the verifier / user can transmit a short (e.g., 128-bit) seed constituting a challenge over an arbitrary number of sentinels; the verifier / user can similarly achieve a high level of assurance on receiving a relatively compact (e.g., 128-bit) proof from the archive. Additionally, the user need not generate sentinel positions and values in a truly random manner, but may instead generate them pseudorandomly using a symmetric key κ. In this case, the user need store only κ and a counter (some tens of bits) in order to perform the POR protocol.

Perhaps the most resource-intensive part of our protocols in practice is the permutation step: This operation requires a large number of random accesses, which can be slow for a file stored on disk (but less so for random-access memory). Our POR construction requires only a single permutation pass, however, and it is possible in some measure

to batch file accesses, that is, to precompute a sequence of accesses and partition them into localized groups. Such detailed questions of system efficiency lie outside the scope of our investigation here.

3.2 Sentinel scheme details

We employ an l-bit *block* as the basic unit of storage in our scheme. We employ an error-correcting code that operates over l-bit symbols, a cipher that operates on l-bit blocks, and sentinels of l bits in length. While not required for our scheme, this choice of uniform parameterization has the benefit of conceptual simplicity. It is also viable in practice, as we demonstrate in our example parameter selections in section 3.4. We also assume for simplicity the use of an efficient (n, k, d)-error correcting code with even-valued d, and thus the ability to correct up to $d/2$ errors.

Suppose that the file F comprises b blocks, $F[1], \ldots, F[b]$. (For simplicity, we assume that b is a multiple of k, a coding parameter. In practice, we can pad out F if needed.) We also assume throughout that F contains a message-authentication code (MAC) value that allows the verifier / user to determine if it has recovered F correctly.

The function encode entails four steps:

1. **Error correction:** We carve our file F into k-block "chunks." To each chunk we apply an (n, k, d)-error correcting code C over $GF[2^l]$. This operation expands each chunk into n blocks and therefore yields a file $F' = F'[1], \ldots, F'[b']$, with $b' = bn/k$ blocks.

2. **Encryption:** We apply a symmetric-key cipher E to F', yielding file F''. Our protocols require the ability to decrypt data blocks in isolation, as our aim is to recover F even when the archive deletes or corrupts blocks. Thus we require that the cipher E operate independently on plaintext blocks. One option is to use a l-bit block cipher. In this case, we require indistinguishability under a chosen-plaintext attack; it would be undesirable, for example, if an adversary in a position to influence F were able to distinguish the data contents of blocks.[2] In practice, an appropriate choice of cipher E would be a tweakable block cipher [24] such as XEX [31]. A second option is to employ a stream cipher E. On decryption, portions of the keystream corresponding to missing blocks may simply be discarded.

3. **Sentinel creation:** Let $f : \{0,1\}^j \times \{0,1\}^* \rightarrow \{0,1\}^l$ be a suitable one-way function (modelled in our proofs by a random oracle). We compute a set of s sentinels $\{a_w\}_{w=1}^s$ as $a_w = f(\kappa, w)$. We append these sentinels to F'', yielding F'''.

4. **Permutation:** Let $g : \{0,1\}^j \times \{1, \ldots, b' + s\} \rightarrow \{1, \ldots, b' + s\}$ be a pseudorandom permutation (PRP) [25]. We apply g to permute the blocks of F''', yielding the output file $\tilde{F}$. In particular, we let $\tilde{F}[i] = F'''[g(\kappa, i)]$.

The function extract requests as many blocks of $\tilde{F}$ as possible. It then reverses the operations of encode. In particular, it decrypts ciphertext blocks, permutes the resulting plaintext blocks under g^{-1}, strips away sentinels, and then applies error correction as needed to recover the original file F. Note that if the code C is *systematic*, i.e., a code word consists of the message followed by error-correction data, then error-correcting decoding is unnecessary when the archive provides an intact file.

To bolster the success probability of extract against probabilistic adversaries, i.e., adversaries that do not respond deterministically to a given challenge value, we do the following. If simple recovery fails, then extract makes an additional $\gamma - 1$ queries for each block and attempts to perform majority decoding over the resulting responses. Given sufficiently large γ, this approach recovers a given block with high probability provided that the adversary outputs a correct response with probability non-negligibly greater than 1/2. We employ this aspect of extract in our security proof.

The function challenge takes as input state variable σ, a counter initially set to 0. It outputs the position of the σ^{th} sentinel by reference to g, i.e., it outputs $p = g(b' + \sigma)$ and increments σ; it repeats this process q times, i.e., generates positions for q different sentinels. The prover function respond takes as input a single challenge consisting of a set of q positions, determines the values of the q corresponding blocks (sentinels in this case), and returns the values. (See below for some simple bandwidth optimizations.) The function verify works in the obvious manner, taking a challenge pair (σ, d) as input and verifying that the prover has returned the correct corresponding sentinels values.[3]

3.2.1 Permutation

The permutation step in our protocol serves two purposes. First, it randomizes the placement of sentinels such that they can be located in constant time and storage; only the sentinel generation key need be stored. Although it is possible instead to insert sentinels into pseudorandomly determined positions, *locating* sentinels placed by straightforward pseudorandom insertion is an operation with time or storage costs linear in the number of sentinels: A lookup procedure must keep track of or reconstruct how other sentinel positions have shifted file contents.

The second purpose relates to error correction. In principle, we could treat our entire file as a single message in an error-correcting code with a large minimum distance, e.g., a Reed-Solomon code. In practice, however, such coding can be challenging—even for erasure-coding. (See [13] on a recent effort to scale a Tornado code to large block sizes.) It is for this reason that we consider the carving of our file F into "chunks." It is important to disperse the constituent blocks of these chunks in a secret, random manner. An adversary with knowledge of the location in $\tilde{F}$ of the blocks belonging to a particular chunk could excise the chunk without touching any sentinels, thereby defeating our POR scheme.

While pseudorandom-permutation primitives are most often designed to operate over bitstrings, and thus power-of-two-sized domains, Black and Rogaway [5] describe simple and efficient pseudorandom-permutation constructions over domains $\mathbb{Z}_k$ for arbitrary integers k. Their constructions are suitable for use in our POR scheme.

[2] In the case of re-use of a file-encryption key, which we deprecate here, it might be necessary to enforce security against chosen ciphertext attacks.

[3] Of course, it is possible for the verifier to pick σ at random from $\{1, \ldots, s\}$, rather than storing it as a counter value. In this case, by the Birthday Paradox, the power of the verifier degrades as the number of used sentinels approaches $\sqrt{s}$.

We note that by treating the file F as a single message and encoding it as a single codeword in a code with large minimal distance, we can in principle eliminate the need to compensate for chunking and thus the one of the requirements for the permutation step in our POR construction. For erasure codes—and thus against an "erasing" adversary—this possibility seems to hold the most practical promise [13]. We consider this possibility in our security analysis below.

3.2.2 Bandwidth optimization

For most practical parameterizations, the verifier will wish to request a batch of q sentinels in a given session. As a simple optimization to reduce bandwidth, the position for each batch of sentinels may be derived from an individual secret key derived from a master key $\kappa_{sentpos}$. The verifier can release the key corresponding to the positions of a given sentinel batch while keeping $\kappa_{sentpos}$ private.

Similarly, rather than returning the full batch of q sentinels, the prover can return a hash or XOR of the sentinels. Thus, the prover's response can consist of a single block, irrespective of the size of q.

An interesting feature of XORing is that it can be employed for compression in a *hierarchical* POR setting. Suppose that an archive A breaks $\tilde{F}$ into pieces and distributes the pieces among a collection of subordinate archives $\{A_i\}$. On receiving a set of challenges, A can parcel them out appropriately to the $\{A_i\}$. Each A_i can return an XOR of its respective responses, which A itself can them XOR together as its response to the prover. This process is transparent to the prover. (And of course, the process can operate recursively over a *tree* of archives.)

3.3 Security

We formally analyze the security of Sentinel-PORSYS$[\pi]$, our sentinel-based POR, in appendix A. Let $C = b'/n$ be the number of constituent chunks (which include data, not sentinels). We define ϵ to be an upper bound on the fraction of data blocks and previously unused sentinels corrupted by the adversary. The total number of such blocks is at most $b' + s$, and decreases over time as sentinels are consumed in verifier challenges. As may be expected, the security of our POR system depends on q, the number of sentinels per challenge, not the total number of available sentinels. In a simplified model that assumes ideal properties for our underlying cryptographic primitives (with little impact on practical parameterization), we prove the following:

THEOREM 1. *Suppose that $\gamma = \omega(\log j) \log b'$. For any probability $\epsilon \in (0, 1)$, then,* Sentinel-PORSYS$[\pi]$ *is a (ρ, λ)-valid POR for $\rho \leq Ce^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2}$ and $\lambda \geq (1 - \epsilon/4)^q$, where $\mu = \frac{n\epsilon}{1-\epsilon}$.*

Some explanation of this security bound is in order. Recall that d is the minimum distance of our error-correcting code; that is, the code can correct at least $d/2$ errors.

As a technical aspect of our proof, we consider a block to be "corrupted" if $\mathcal{A}$("respond") returns it correctly with probability less than 3/4. (The constant 3/4 is arbitrary; our proofs work for any constant greater than 1/2, with changes to the constants in our theorem.) Recall that our security definition for a POR treats the extraction probability $1 - \zeta$ in an asymptotic sense for the sake of simplicity. We analyze γ—the number of queries made by extract on a given block—accordingly. Given the lower bound $\gamma = \omega(\log j) \log b'$, we can show that the verifier recovers all uncorrupted blocks from $\mathcal{A}$("respond") with overwhelming probability.[4]

Given the ability to recover all uncorrupted blocks, it is possible to recover the full file F provided that the adversary does not corrupt more than $d/2$ blocks in any chunk. In this case, it is possible to recover the data associated with the chunk through error-correction. The value ρ bounds the probability of more than $d/2$ corruptions in any chunk when a random ϵ-fraction of blocks is corrupted for $\epsilon \in (0, 1)$.[5] Naturally, the larger the minimum distance d, the lower the probability of such corruption, as reflected in the theorem bounds.

Our bound for λ simply reflects the probability of an adversary successfully returning q sentinels when it has corrupted an ϵ-fraction of blocks.

As remarked above, some erasure codes, e.g., Raptor codes [2], operate in linear time and may be amenable in some cases to practical application to entire large files without any need for "chunking" [13]. Additionally, it is possible (if not generally practical) to treat a full file as a single message in an error-correcting code with large minimum distance. In such cases, we can obtain considerably tighter bounds on the security of our POR system.

Consider a system Sentinel-PORSYS$[\pi]$ that is: (1) Implemented against an *erasing* adversary without chunking using an erasure code with minimum distance $d + 1$ or (2) Implemented against a fully capable adversary using an error-correcting code with minimum distance $2d$ and no chunking. In both cases, if $\epsilon \leq d/b'$, then the file F is fully recoverable. Additionally, we make the following observation (whose proof follows straightforwardly from the analysis underlying our main theorem):

OBSERVATION 1. *Suppose that $\epsilon > d/b'$. Then* Sentinel-PORSYS$[\pi]$ *is a (ρ, λ)-valid POR for $\rho = 0$ and $\lambda \geq (1 - \epsilon/4)^q)$.*

3.4 An example parameterization

A block size of $l = 128$ is one natural choice; 128 bits is the size of an AES block and yields sentinels of sufficient size to protect against brute-force sentinel-guessing attacks. Let us consider use of the common $(255, 223, 32)$-Reed-Solomon code over $GF[2^8]$, i.e., with one-byte symbols. By means of the standard technique of "striping" (see, e.g., [7]), we can obtain a $(255, 223, 32)$-code over $GF[2^{128}]$, i.e., over file blocks, which is convenient for our parameterization in this example. A chunk consists then of $n = 255$ blocks.

Let us consider a file F with $b = 2^{27}$ blocks, i.e., a 2-gigabyte file. This file expands by just over 14% under error-coding to a size of $b' = 153,477,870$. Suppose that we add $s = 1,000,000$ sentinels. Thus the total number of data blocks $b' + s = 154,477,870$, the total number of blocks in the file $\tilde{F}$. The total file expansion is around 15%.

Consider $\epsilon = 0.01$, i.e., an adversary that has corrupted up to 1% of the data blocks and unused sentinels in $\tilde{F}$. Now

[4]Since extract needs to perform multiple queries *only* in the presumably rare case of a file-retrieval failure against a probabilistic adversary, we can make γ large in most practical settings, as when an adversarial archive is taken offline and "rewound" to extract block values.

[5]While excluded for technical reasons, we can see trivially that for $\epsilon = 0$, F is retrievable with probability 1. When $\epsilon = 1$, the theorem holds for $\rho = 0$ and $\lambda \geq (1 - \epsilon/4)^q$.

$C = b'/n = 601,874$, and $\mu = \frac{n\epsilon}{1-\epsilon} = 2.58$. (Recall μ is an upper bound on the mean number of corrupted blocks per chunk.) By Theorem 1, $\rho \leq Ce^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2} = 601,874 \times e^{-23.26} \approx 4.8 \times 10^{-5}$. In other words, the probability that the adversary renders the file unretrievable[6] is about 1 in 20,000.

Suppose that we let $q = 1,000$, i.e., the verifier queries 1,000 sentinels with each challenge. (Recall that XORing or hashing compresses a response into 128 bits.) Since the total number of sentinels is $s = 1,000,000$, the verifier can make 1,000 challenges over the life of the file (a challenge per day for about three years). The probability of detecting adversarial corruption of the file is $1-\lambda = 1-(1-\epsilon/4)^q \approx 91.8\%$ *per challenge*. This is not overwhelmingly large, but probably suffices for most purposes, as detection of file-corruption is a cumulative process.

Of course, for higher ϵ, the probability of file corruption is higher, but so too is that of detection by the verifier.

3.5 Variant protocols

To shed light on the architectural choices in our protocol, it is helpful to make brief mention of a couple of simple variants on our POR scheme Sentinel-PORSYS.

3.5.1 Authenticated blocks

The role of the sentinels in Sentinel-PORSYS is to permit detection of file corruption. It is possible, however, to achieve this goal by direct verification of file contents. For example, we might omit sentinels entirely and instead use message-authentication codes (MACs) to verify the correctness of file contents. In this case, we would omit the sentinel insertion step from Sentinel-PORSYS, and add the following, final step after permutation. We partition the file $\tilde{F}$ into sequentially indexed segments of v blocks for appropriate parameter v (say, 5). To each segment we append a MAC under key κ of the contents of the blocks within the segment, the segment index, and the file handle. For the purposes here, the bit-length of a MAC can be small, e.g., 20 bits, since it is the aggregate effect on multiple blocks that is being measured. Thus, the resulting file expansion need not be prohibitive. For example, with 20-byte blocks, $v = 5$, and 20-bit MACs, the incremental file expansion due to MACing would be only 2.5%.

There is a straightforward challenge-response procedure in this variant of Sentinel-PORSYS. The verifier picks a segment index at random, asks the prover to return the corresponding segment and MAC, and then verifies the MAC.

This MAC-based approach is quite efficient in terms of file-expansion overhead, computational costs, and bandwidth. It has an important drawback, though: It does not permit the prover to return a digest of its reponses, i.e., to hash or XOR them together. Our MAC-based variant does have the interesting feature, though, of permitting a model in which challenges derive from a (fresh) common reference string or public source of randomness. (Another possible but less efficient way to verify the correctness of file blocks is use of a Merkle tree in which each segment corresponds to a leaf, and the verifier stores the root.)

3.5.2 PIR schemes

Another variant on Sentinel-PORSYS involves the use of *private information retrieval* (PIR) [21]. In explicitly requesting the value of a sentinel from the prover, the verifier discloses the location of the sentinel. A PIR scheme, in contrast, permits a verifier to retrieve a portion of a file $\tilde{F}$ from a prover without disclosing what it has retrieved. Thus, by retrieving sentinels using PIR, the verifier can re-use sentinels, i.e., let $q = s$, with no effective degradation in security.

While capable of communication costs of $O(\log^2(|\tilde{F}|))$ per retrieved sentinel bit [23], PIR schemes require access to all of $\tilde{F}$ and carry high computational costs. Recent work suggests that PIR schemes may be no more efficient in practice than transferring all of $\tilde{F}$ [33].

4. PRACTICAL APPLICATION TO STORAGE SERVICES

We now describe an application of our POR scheme to an archive or back-up service provided by a Storage Service Provider (SSP). Multiple service levels may be offered, corresponding to different storage "tiers" (disks of varying speeds, tape, etc.) or a combination (see, e.g., [32]). An SSP and a client typically operate under a *service level agreement* (SLA) specifying properties such as throughput, response time, availability, and recovery-time objectives [28].

The price of the service is set by the SSP at some profit margin above the cost of providing the service at a given level (equipment, maintenance, staff, facilities, etc.). An SSP is thus motivated legitimately to increase its profit margin by reducing cost while maintaining the same service level; in a competitive marketplace this will ultimately reduce the price, which is a benefit to clients as well. (Indeed, one of the reasons for outsourcing to an SSP is the clients belief that the SSP can reduce the price more effectively than the client alone.)

Internet Service Providers (ISPs) and Application Service Providers (ASPs) follow a similar economic model, but with an important difference. An ISP or ASPs service levels are effectively tested continuously by clients for most functions as a part of their regular interactions with the services. An SSP's service levels, on the other hand, particularly for its restore and retrieve services (corresponding respectively to backup and archive functions) are only tested when those functions are actually run, which in general is infrequent.[7] Furthermore, the situations in which those functions are run are the very ones for which the client is paying for service levels: The client does not (in general) pay a higher price to have backup or archive data stored faster, but rather to ensure that it can be accessed faster.

Without a requirement to provide continuous service level assurances, an SSP may also be willing to take the risk of decreasing its cost by not maintaining an agreed service level. For instance, an SSP may place files it considers less likely to be accessed to a lower tier of storage than agreed. Or, an SSP may *not* move files to higher speed disks when they become available, despite agreeing to do so. These lapses are exacerbated by the possibility that the SSP may itself rely

[6]We assume negligible ζ in this example. It may be seen, for instance, that $\gamma = 1800$ yields an extraction failure probability $\zeta < 2^{-80}$.

[7]In the general case where an SSP also provides primary read/write file storage, the service levels would be tested continuously, at least for those primary files that are accessed frequently. Here, the focus is on secondary storage with less frequent reads.

on other SSPs to store files or parts of them. For instance, to meet an agreed availability level, an SSP may replicate or otherwise store data on geographically distributed sites, perhaps employing information dispersal techniques as suggested in section 1.3. Some of these sites may be operated by other SSPs, who in turn may have their own motivations to reduce cost, legitimate or otherwise. If a site knows that its occasional outage will be overlooked (indeed, planned for) due to the presence of its replication peers, it may opt to increase its frequency of "outages" by placing a fraction of files on lower tiers—or not storing them at all.

Cost reduction is one reason an SSP might not store files as directed. Another relates to the information in the files themselves. A malicious SSP may also wish for certain files (or portions of them) not to be accessible due to their content. Encryption partially mitigates this threat since an SSP does not directly know the content of encrypted files, but it may still be possible for other parties to inform the SSP by back channels of which files to "misplace" or to cause the misplacement themselves by physical attack. E-discovery of documents is one scenario motivating these concerns.

Equipment failures and configuration errors may also result in file placement that does not meet an SLA; the breach of agreement may simply be due to negligence, not malice.

One way for a client to obtain assurance that a file can be accessed at a given service level, of course, is for the client actually to access the file from time to time. Indeed, file access is part of a typical "fire drill" operation for disaster recovery testing. Such spot checks may also be beneficial for operational recovery. If randomly chosen files are accessible at a given service level, then it is reasonable to assume that other files will be accessible as well. However, the highest assurance for a specific file requires access to the file itself. An occasional full read of the file is one way to obtain this assurance, but at a significant operational cost over the life of the file. The POR schemes presented here aim to provide this assurance less expensively.

We envision that a POR scheme would be applied to backup or archive as follows. As part of its SLA, an SSP would offer periodic, unannounced execution of a POR for selected files. In the POR, a block would be considered to be an erasure if it cannot be read within the agreed response time.[8] The client, taking the role of verifier, would thereby obtain (probabilistic) assurance that the agreed service level continues to be met for the file. If the SSP is trusted to provide file integrity, then an erasure code would be sufficient for error correction.

A POR scheme can also be applied by a third party to obtain assurance that files are accessible. For instance, an auditor may wish to verify that an SSP is meeting its SLAs. To ensure that the POR corresponds to a file actually submitted for storage, the auditor would rely on the client to provide a storage "receipt" including the keys the verification operations. (The key for decrypting the file need not be provided—thus enforcing privilege separation between the auditor and the client.) As another example, one party's legal counsel in a dispute may wish to verify that an archive stored at an SSP correctly corresponds to a document manifest submitted by another party. The separation of encryption and verification enables the legal counsel (and the court) to verify that the other party has met a requirement to archive a collection of files, without yet learning the content of those files—and, due to the POR, without having to access every block in every file.

[8]The agreed response time is a composition of the latency and the throughput—a delay until file access begins, a data rate thereafter. The agreed time for POR acceptance may need to be calculated differently in file systems optimized for data streaming, where the SLA promises a data rate for sequential access but not necessarily for the random access in the POR. Furthermore, if the SLA for the SSP does not directly govern network latencies, then the SLA for the ISP may need to be factored in as well; over open networks, the agreed response time would take into account "expected" behavior, e.g., compared to general network latencies at the time of the proof.

5. CONCLUSION

Thanks to its basis in symmetric-key cryptography and efficient error-coding, we believe that our sentinel-based POR protocol is amenable to real-world application. As storage-as-a-service spreads and users rely on external agents to store critical information, the privacy and integrity guarantees of conventional cryptography will benefit from extension into POR-based assurances around data availability. Contractual and legal protections can, of course, play a valuable role in laying the foundations of secure storage infrastructure. We believe that the technical assurances provided by PORs, however, will permit even more rigorous and dynamic enforcement of service policies and ultimately enable more flexible and cost-effective storage architectures.

Our introduction of PORs in this paper leads to a number of possible directions for future research. One of these, of course, is how to perform the most effective error-coding in our POR protocols. Another broad area of research stems from the fact that our main POR protocol is designed to protect a *static* archived file F. Any naïvely performed, partial updates to F would completely undermine the security guarantees of our protocol. For example, if the verifier were to modify a few data blocks (and accompanying error-correcting blocks), the archive could subsequently change or delete the set of modified blocks with impunity, having learned that they are not sentinels. A natural question then is how to construct a POR that can accommodate partial file updates—perhaps through the dynamic addition of sentinels. A related question arises from the one-time nature of sentinels in our main POR protocol. Our construction assumes an initial embedding in F of sentinels sufficient for a lifetime of use. Of course, sentinels can be updated through wholesale re-coding of F. Similarly, our PIR-variant permits sentinel re-use (and our MAC variant avoids it). A natural open question is whether there is a practical means of improving on one-time sentinel use.

6. REFERENCES

[1] Amazon.com. Amazon simple storage service (Amazon S3), 2007. Referenced 2007 at aws.amazon.com/s3.

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

[3] R. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. In R. Guerraoui, editor, *DISC '04*, pages 405–419. Springer, 2004. LNCS vol. 3274.

[4] M. Bellare and O. Goldreich. On defining proofs of knowledge. In E.F. Brickell, editor, *CRYPTO '92*, pages 390–420. Springer, 1992. LNCS vol. 740.

[5] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In B. Preneel, editor, *CT-RSA '02*, pages 114–130. Springer, 2002. LNCS vol. 2271.

[6] R. Burns. Verifying the preservation of large data sets, 2007. Research update for NSF grant IIS-0456027, "Securely Managing the Lifetime of Versions in Digital Archives." Referenced 2007 at http://hssl.cs.jhu.edu/digarch/nsf.pdp.pdf.

[7] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Reliable Distributed Systems (SRDS) '05*, pages 191–202, 2005.

[8] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *DSN '06*, pages 115–124, 2006.

[9] D. E. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE S & P '05*, pages 139–153, 2005.

[10] B. F. Cooper and H. Garcia-Molina. Peer to peer data trading to preserve information. *ACM Trans. Inf. Syst.*, 20(2):133–170, April 2002.

[11] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In D. Boneh, editor, *CRYPTO '03*, pages 426–444. Springer, 2003. LNCS vol. 2729.

[12] IDC. J. F. Gantz et al. The expanding digital universe: A forecast of worldwide information growth through 2010, March 2007. Whitepaper.

[13] J. Feldman. Using many machines to handle an enormous error-correcting code. In *IEEE Information Theory Workshop (ITW)*, 2006. Referenced 2007 at http://www.columbia.edu/$\mathcal{S}$jf2189/pubs/bigcode.pdf.

[14] D. L. G. Filho and P. S. L. M. Barreto. Demonstrating data possession and uncheatable data transfer, 2006. IACR eArchive 2006/150. Referenced 2007 at http://eprint.iacr.org/2006/150.pdf.

[15] O. Goldreich. *Randomness, interactive proofs, and zero-knowledge–A survey*, pages 377–405. Oxford University Press, 1988.

[16] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM J. Comput*, 18(1):186–208, 1989.

[17] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In M. Blaze, editor, *Financial Cryptography '02*, pages 120–135. Springer, 2002. LNCS vol. 2357.

[18] P. Golle and I. Mironov. Uncheatable distributed computations. In D. Naccache, editor, *CT-RSA '01*, pages 425–440. Springer, 2001. LNCS vol. 2020.

[19] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In B. Preneel, editor, *Communications and Multimedia Security*, pages 258–272. Kluwer, 1999.

[20] V. Kher and Y. Kim. Securing distributed storage: Challenges, techniques, and systems. In *StorageSS 05*, pages 9–25, 2005.

[21] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS '97*, pages 364–373. IEEE Computer Society, 1997.

[22] L. Lamport. On interprocess communication. part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[23] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In J. Zhou and J. Lopez, editors, *Information Security Conference (ISC) '05*, pages 314–328. Springer, 2005. LNCS vol. 3650.

[24] M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In M. Yung, editor, *CRYPTO '02*, pages 31–46. Springer, 2002. LNCS vol. 2442.

[25] M. Luby and C. Rackoff. How to construct pseudorandom permutations and pseudorandom functions. *SIAM J. Comput.*, 17:373–386, 1988.

[26] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[27] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *DISC '02*, pages 311–325. Springer, 2002. LNCS vol. 2508.

[28] E. St. Pierre. ILM: Tiered services and the need for classification. In *Storage Networking World (SNW) '07*, April 2007. Slide presentation.

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

[30] R. Rivest. The pure crypto projects hash function. Cryptography Mailing List Posting. Referenced 2007 at http://diswww.mit.edu/bloom-picayune/crypto/13190.

[31] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In P. J. Lee, editor, *ASIACRYPT '04*, pages 16–31. Springer, 2004. LNCS vol. 3329.

[32] X. Shen, A. Choudhary, C. Matarazzo, and P. Sinha. A multi-storage resource architecture and I/O performance prediction for scientific computing. *J. Cluster Computing*, 6(3):189–200, July 2003.

[33] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Network and Distributed Systems Security Symposium (NDSS) '07*, 2007. To appear.

[34] M. Yung. Zero-knowledge proofs of computational power (extended summary). In J.J. Quisquater and J. Vandewalle, editors, *EUROCRYPT '89*, pages 196–207. Springer, 1989. LNCS vol. 434.

APPENDIX

A. PROOFS

For simplicity, we make ideal assumptions on our underlying cryptographic primitives. We assume an ideal cipher, a one-way permutation instantiated as a truly random permutation, and a PRNG instantiated with truly random outputs. Given well-constructed primitives, these ideal assumptions should not impact our security analysis in a practical sense. Viewed another way, we assume parameterizations of our cryptographic primitives such that the probability of an adversary distinguishing their outputs from suitable random distributions is negligible. The error terms in our theorems should therefore be small. (In a full-blown proof, we would create a series of games/simulators that replace primitives incrementally with random distributions.)

This ideal view yields a system in which block values are distributed uniformly at random in the view of the adversary. Thus the adversary cannot distinguish between blocks corresponding to message values and those corresponding to sentinels, and cannot determine which blocks are grouped in chunks more effectively than by guessing at random.

Additionally, in this model, because $\mathcal{A}$("respond") is assumed to be stateless, i.e., subject to rewinding, the verifier can make an arbitrary number of queries on a given block. These queries are independent events, i.e., for a given block i, the probability $p(i)$ that $\mathcal{A}$("respond") responds correctly is equal across queries. Hence we can model the adversary $\mathcal{A}$("respond") as a probability distribution $\{p(i)\}$ over blocks in the archived file.

Furthermore, if $p(i)$ is non-negligibly greater than $1/2$ (as a technicality in our proof, if $p(i) \geq 3/4$), it is possible for an extractor to recover block i with overwhelming probability via majority decoding after a small number ($\gamma = \omega(\log j) \log b'$) of queries. Thus, we can simplify our model still further. Once we have bounded out the probability of the adversary failing to retrieve a block i with $p(i) \geq 3/4$, we may think effectively of $\mathcal{A}$("respond") in our ideal model as a collection of "bins," each corresponding to a given block i. The adversary is modelled as corrupting a block by throwing a "ball" into the corresponding bin. If a bin contains a ball, then we assume $p(i) < 3/4$, and thus that the corresponding block is not retrievable. If a bin doesn't contain a ball, the corresponding block is retrievable.

Let us define b'' as the total number of data blocks and previously unused sentinels; thus, $b' + q \leq b'' \leq b' + s$. Recall that we define ϵ as the fraction of the b'' such blocks corrupted by the adversary. In the ball-and-bin view, therefore, the adversary throws a number of balls $\epsilon b''$ into the b'' bins representing this aggregate of blocks. Provided that no chunk receives too many balls, i.e., provided that no chunk of n error-correcting blocks in a codeword has more than $d/2$ corruptions, extract can recover the file F completely. We use the ball-and-bin model to achieve bounds on the probability of success of extract. We also use the ball-and-bin model to bound the probability of detecting adversarial corruption of F.

As the only adversarial function we refer to explicitly in our proofs is $\mathcal{A}$("respond"), we write $\mathcal{A}$ for conciseness.

A.1 Bounding lemmas

We begin with some technical lemmas to establish bounds within our ball-and-bin model of $\mathcal{A}$("respond"). The lynchpin of these lemmas is the well-known Chernoff probability bounds on independent Bernoulli random variables, as expressed in the following lemma.

LEMMA 1 (CHERNOFF BOUNDS). *Let $X_1, X_2, \ldots, X_N$ be independent Bernoulli random variables with $\mathsf{pr}[X_i = 1] = p$. Then for $X = \sum_{i=1}^{N} X_i$ and $\mu = E[X] = pN$, and any $\delta \in (0, 1]$, it is the case that $\mathsf{pr}[X < (1-\delta)\mu] < e^{-\mu\delta^2/2}$ and $\mathsf{pr}[X > (1+\delta)\mu] < e^{-\mu\delta^2/3}$.*

Now let us consider a simple algorithm that we refer to as a *γ-query majority decoder*. It operates in two steps: (1) The decoder queries $\mathcal{A}$ on a given block i a total of γ times, receiving a set R of responses and then (2) If there exists a majority value $r \in R$, the decoder outputs r; otherwise it outputs $\perp$.

We state the following lemma without proof, as it follows straightforwardly from Lemma 1.

LEMMA 2. *The probability that a γ-query majority decoder operating over b' blocks correctly outputs every block i for which $p(i) \geq 3/4$ is at least $1 - b'e^{-3\gamma/72}$.*

Lemma 2 is a bound on the probability that the ball-and-bin model correctly models the behavior of $\mathcal{A}$. Our next lemma operates within the ball-and-bin model. It bounds the probability, given $\epsilon b''$ bins with balls, i.e., corrupted blocks, that any chunk is corrupted irretrievably.

LEMMA 3. *Suppose $\epsilon b''$ (for $\epsilon \in [0, 1]$) balls are thrown into b'' bins without duplication, i.e., with at most one ball per bin. Suppose further that a subset of $b' < b''$ bins is partitioned randomly into $C = b'/n$ chunks, each comprising n distinct bins. Let $\mu = \frac{n\epsilon}{1-\epsilon}$. Then the probability that any chunk receives more than $d/2$ balls is bounded above by $Ce^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2}$.*

Proof: Since balls are thrown without duplication, the maximum probability that a given chunk receives a ball is achieved under the condition that $\epsilon b'' - 1$ balls have already landed outside the chunk. Thus, the conditional probability that a ball lands in a given chunk is bounded above by $p = n/(b'' - \epsilon b'')$.

The number of balls that lands in a given chunk is therefore bounded above by a Bernoulli process in which X_i is the event that the i^{th} ball lands in the chunk, $p_i = p$, and $X = \sum_{i=1}^{\epsilon b''} X_i$. We have $E[X] = \mu = p\epsilon b'' = n\epsilon/(1-\epsilon)$.

Now $\mathsf{pr}[X > d/2] = \mathsf{pr}[X > (1+\delta)\mu]$ for $\delta = \frac{d}{2\mu} - 1$. Thus, by Lemma 1, we have $\mathsf{pr}[X > d/2] < e^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2}$. Since there are C chunks, the lemma follows. ∎

Our next lemma offers a lower bound on the probability that the verifier detects file-corruption by $\mathcal{A}$ when at least an ϵ-fraction of bins contain balls, i.e., $\mathcal{A}$ responds incorrectly with probability at least $1/4$ on an ϵ-fraction of data blocks and unused sentinels.

LEMMA 4. *Suppose that $\epsilon b''$ (for $\epsilon \in [0, 1]$) balls are thrown into b'' bins without duplication. Suppose that $\mathcal{A}$ is queried on q bins, each chosen uniformly at random (and independently). The probability that $\mathcal{A}$ provides at least one incorrect block is at least $1 - (1 - \epsilon/4)^q$.*

Proof: Let X_i be a Bernoulli random variable s.t. $X_i = 1$ if $\mathcal{A}$ provides an incorrect block on the i^{th} query. Since a bin contains a ball with probability ϵ, and a bin with a ball corresponds to a block incorrectly emitted by $\mathcal{A}$ with probability at least $1/4$, $\mathsf{pr}[X_i = 0] \leq 1 - \epsilon/4$. It is easy to see that $\mathsf{pr}[X_i = 0 \mid X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_q = 0] \leq \mathsf{pr}[X_i = 0]$. Therefore $\mathsf{pr}[X = 0] \leq (1 - \epsilon/4)^q$. ∎

A.2 Main theorem

We state the following theorem about our sentinel-based practical scheme Sentinel-PORSYS$[\pi]$:

THEOREM 1. *Suppose that $\gamma = \omega(\log j) \log b'$. For any $\epsilon \in (0, 1)$, then, Sentinel-PORSYS$[\pi]$ is a (ρ, λ)-valid POR for $\rho \leq Ce^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2}$ and $\lambda \geq (1 - \epsilon/4)^q$, where $\mu = \frac{n\epsilon}{1-\epsilon}$.*

Proof: Consider a given value of ϵ in our balls-and-bins model. By Lemma 4, the probability for this value of ϵ that $\mathcal{A}$ causes the the verifier to accept is bounded above by $\lambda = (1 - \epsilon/4)^q$. (To achieve a higher verfier acceptance probability than λ would require a larger value of ϵ.)

Given this value of ϵ, by Lemma 2, the probability of recovering correct values for all blocks that have not received balls is at least $1-\zeta$ for $\zeta = b'e^{-3\gamma/72}$. For $\gamma = \omega(\log j)\log b'$, we have $\zeta = j^{-\omega(1)}$, which is negligible in j.

Assuming, then that no "chunk" has received more than $d/2$ balls, the verifier can recover data associated with every chunk of the file. By Lemma 3, this condition fails to hold with probability at most $Ce^{-\frac{\mu}{3}(\frac{d}{2\mu}-1)^2}$. ∎