Proofs of Retrievability: Theory and Implementation

Kevin D. Bowers, Ari Juels, and Alina Oprea
RSA Laboratories, Cambridge, MA

ABSTRACT

A *proof of retrievability* (POR) is a compact proof by a file system (prover) to a client (verifier) that a target file F is intact, in the sense that the client can fully recover it. As PORs incur lower communication complexity than transmission of F itself, they are an attractive building block for high-assurance remote storage systems.

In this paper, we propose a theoretical framework for the design of PORs. Our framework improves the previously proposed POR constructions of Juels-Kaliski and Shacham-Waters, and also sheds light on the conceptual limitations of previous theoretical models for PORs. It supports a fully Byzantine adversarial model, carrying only the restriction—fundamental to all PORs—that the adversary's error rate ϵ be bounded when the client seeks to extract F. Our techniques support efficient protocols across the full possible range of ϵ, up to ϵ non-negligibly close to 1.

We propose a new variant on the Juels-Kaliski protocol and describe a prototype implementation. We demonstrate practical encoding even for files F whose size exceeds that of client main memory.

1. INTRODUCTION

Cloud computing, the trend toward loosely coupled networking of computing resources, is unmooring data from local storage platforms. Users today regularly access files without knowing on what machines or in what geographical locations their files reside. They may even store files on platforms with unknown owners and operators, particularly in peer-to-peer computing environments.

While cloud computing encompasses the full spectrum of computing resources, in this paper we focus on *archival* or *backup* data, large files subject to infrequent updates. While users may access such files only sporadically, a demonstrable level of availability may be required contractually or by regulation. Financial records, for instance, have to be retained for several years to comply with recently enacted regulations.

Juels and Kaliski (JK) recently proposed a notion for archived files that they call a *proof of retrievability* (POR). A POR is a protocol in which a server/archive proves to a client that a target file F is intact, in the sense that the client can retrieve all of F from the server with high probability. In a naïve POR, a client might simply download F itself and check an accompanying digital signature. JK and related constructions adopt a challenge-response format that achieves much lower (nearly constant) communication complexity—as little as tens of bytes per round in practice.

JK offer a formal definition of a POR and describe a set of different POR designs in which a client stores just a single symmetric key and a counter. Their most practical constructions, though, support only a limited number of POR challenges. Shacham and Waters (SW) offer an alternative construction based on the idea of storing homomorphic block integrity values that can be aggregated to reduce the communication complexity of a proof. Its main advantage is that, due to the underlying block integrity structure, clients can initiate and verify an unlimited number of challenges.

In this paper, we introduce a general conceptual framework for PORs. The resulting design space encompasses both JK and SW, and leads naturally to variants—and improvements—on both proposals. We examine in particular a variant of JK that simultaneously achieves lower storage requirements and a higher level of assurance than JK, with minimal computational overhead. We describe a prototype implementation of this improved scheme. While SW has strong theoretical advantages over JK, e.g, it can tolerate any adversarial error rate $\epsilon < 1$, and can support arbitrarily many POR interactions, we focus on JK in this paper because of its advantages in a practical setting. Although our framework supports efficient protocols up to $\epsilon < 1$ and is applicable to SW, we focus our implementation on a variant of the JK protocol requiring $\epsilon < \frac{1}{2}$ and supporting only a bounded number of queries. As we explain, JK's theoretical restrictions are of limited impact in practical deployment scenarios. (For example, an adversary with $\epsilon > \frac{1}{2}$ would be swiftly detected by a client's POR challenges.) As we show in Section 5, JK carries lower resource costs than SW in many settings of real-world interest.

1.1 Intuition for our framework

In a POR protocol, a file is encoded by a client before being transmitted to a storage provider for archiving. A POR enables bandwidth-efficient challenge-response protocols to guarantee probabilistically that a file is available at a remote storage provider. Most PORs proposed to date, and the ones we consider in this paper, use the technique of "spot-checking" in the challenge-response protocol to detect adversarial behavior. In each challenge, a subset of file blocks is sampled, and the results of a computation over these blocks is returned to the client. The returned results are checked using some additional information embedded into the file at encoding time.

Our POR framework operates in the strongest possible adversarial model: it assumes that the adversary replies correctly

to a fraction of challenges $1 - \epsilon > 0$ chosen uniformly at random from the challenge space (we call such an adversary an ϵ-adversary). The technique of spot-checking can only detect a large adversarial corruption rate. For POR protocols to be resilient to small adversarial corruption, an error-correcting code (called *outer code*) is used to encode the full file.

By analogy with zero-knowledge proofs, the same interface used for challenge-response interactions between the client and server is also available for *extraction*. The client first tries to download F as normal (checking the integrity of the file against a MAC or digital signature of the file). If this usual process fails, then the client resorts to a POR-based extraction. The client in this case submits challenges to the server and reconstructs F from the (partially corrupted) values yielded by the server via respond.

In most previously proposed POR constructions, the function respond returns a single file block or an XOR of file blocks. Our key insight in this paper is a technique that bases respond itself on an *arbitrary error correcting code*. In particular, we consider schemes in which respond computes a codeword on the blocks in s and returns the uth symbol. We refer to this as the *inner code* ECC_{in} and to the code ECC_{out} as the *outer code*.[1]

The concatenation or interleaving of two or more error-correcting codes is a classical technique for creating error-correcting codes with certain properties, e.g., resilience to burst errors. The way we compose the inner and outer codes in our construction is similar in spirit, but quite different in its application, which is aimed specially at PORs. The two codes play complementary roles, but operate in distinct ways and at different protocol layers. Most notably, while the inner code may vary dynamically based according to the challenge format, the outer code serves in the initial file encoding, and is thus static over the lifetime of a file.

The inner code ECC_{in}, being computed on the fly by the server over the encoded file $\tilde{F}$, creates no storage overhead. On the other hand, it imposes a computational burden on the server when it responds to client challenges: The server must retrieve the blocks in s and apply the code ECC_{in} to them. The outer code imposes little computational burden for the server, but results in an expansion of the stored file: The greater the error tolerance of ECC_{out}, the larger $|\tilde{F}|/|F|$. In designing a practical POR, we seek to strike a good balance in our selection of ECC_{out} and ECC_{in}.

POR protocols designed in our framework need to specify a maximum bound on the adversarial corruption rate ϵ they tolerate. Our POR framework proceeds in two phases. In Phase I, the client performs a series of challenge-response interactions with the server with the aim of detecting if the adversary is respecting the ϵ bound on corruption rate. (Phase I is in principle optional if we assume an adversary with error rate below ϵ, but helps guarantee file-retrieval efficiency.) In Phase II, the client extracts the file, assuming that it deals with an ϵ-adversary.

Based on the number of verifications they support over the lifetime of the system, we can classify PORs into two main types:

1. PORs that enable *unlimited* number of verifications, such as the SW scheme, are usually constructed by storing

additional integrity values for file blocks. SW employs homomorphic integrity values (essentially MACs) that can be aggregated over multiple blocks, resulting in a bandwidth-efficient POR. In this case, thanks to the underlying system of MACs, the inner code can simply be an erasure code and the scheme can tolerate error rates ϵ non-negligibly close to 1 (albeit at the cost of extraction polynomial in $\frac{1}{1-\epsilon}$). We describe in Section 4 a generalization of SW that fits into our framework and show how we can improve and simplify the SW construction. Theoretically, such protocols do not need to employ Phase I to limit ϵ (since they can extract for ϵ close to 1). In practice, it is still essential to bound the adversarial corruption rate using a Phase-I protocol, since this ensures that a file can be extracted *efficiently*.

2. PORs that can verify a *limited* number of queries, such as the JK scheme, usually pre-compute the responses to a set of challenges and embed them (encrypted) into the file encoding. The verification capability of these PORs is limited by the number of pre-computed challenges embedded into the file encoding. These types of POR cannot feasibly check all the responses received during Phase II, and thus need to employ error-correcting inner codes. In such protocols, we require an upper bound on the adversarial rate ϵ less than $\frac{1}{2}$; the actually tolerable ϵ depends on the inner and outer code parameters. For such protocols to fit our framework, we need to assume that the adversary's corruption rate in Phase I is also limited by the same ϵ determined by the extraction capability in Phase II. (The same assumption is required for SW if we are to guarantee efficient extraction.) Under this assumption, either an adversary has corruption rate greater than ϵ, in which case it will be detected in Phase I, or ϵ is low enough so that it is possible to extract the file in Phase II.

In Section 4 we present a more efficient version of JK that employs a limited number of verifications assuming an upper bound on ϵ given by the error correction rates of the inner and outer codes. Despite such restrictions, in Section 5 we show its practical advantages over SW, in terms of storage overhead and proof costs, for particular ranges of ϵ.

Besides providing a theoretical framework for designing POR protocols and showing improvements to existing protocols, in this paper we also consider the challenges encountered when designing *practical* POR protocols. First, we show how to construct outer codes that can encode large files efficiently, while still preserving a high minimum distance. We define and construct practical *adversarial error-correcting codes* that, intuitively, give no advantage to an adversary in corrupting the encoded file than distributing corruptions randomly across file blocks. Secondly, as random disk access is expensive, we present techniques to encode large files *incrementally*, in only *one pass* through the file.

1.2 Organization

In Section 2, we review existing research related to PORs. We describe our proposed conceptual framework in Section 3. After presenting the full details of a new variant POR scheme and its security proof in Section 4, we compare it with the

---

[1]"Inner code" and "outer code" are the standard names for the constituents of a *concatenated* error-correcting code [20].

original JK protocol and with the SW scheme in Section 5. Finally, we describe several challenges we encountered in implementing the new variant and present performance evaluation in Section 6. We defer the security analysis of the proposed construction in the full version of the paper [6].

2. RELATED WORK

The first proposed POR-like construction of which we are aware is that of Lillibridge et al. [14]. Theirs is a distributed scheme in which blocks of a file F are dispersed in shares across n servers using an (m, n)-erasure code. Servers perform spot checks on the integrity of one another's fragments using message authentication codes (MACs). These MACs also have the effect of allowing reconstruction of F in the face of data corruption, i.e., turning the erasure code into an error-correcting code. Corrupted blocks are discarded, and thus treated as erasures. Lillibridge et al. do not offer formal definitions or analysis of their scheme.

Naor and Rothblum [18], extending the memory-checking schemes of Blum et al. [5], describe a theoretical model that may be viewed as a generalization of PORs. Their model supposes the publication of a (potentially corrupted) encoded file $\tilde{F}$, meaning that the client can directly sample segments of $\tilde{F}$. In the NR construction, $\tilde{F}$ includes message authentication codes (MACs) on file blocks: A client can check the intactness of a file by verifying the correctness of randomly sampled file blocks. As in Lillibridge, an error-correcting code ensures file recovery in the face of some degree of file corruption. NR, however, do not naturally model PORs with nontrivial challenge-response protocols, as required for our purposes in this paper.[2] Additionally, NR propose a construction in which the client applies a high minimum-distance error correcting code across all of F. As we explain below, one of the important challenges in practical schemes is the negotiation of complicated error-coding strategies; a code across all of F is not necessarily practical.

Juels and Kaliski [13] propose a formal POR protocol definition and accompanying security definitions which we describe below. As in NR, they propose a scheme in which the client applies an error-correcting code to file F to obtain the (expanded) file $\tilde{F}$ stored on the server. JK do not store MACs for individual file blocks. Instead, the client challenges the server by specifying a subset of file blocks s_i from a predetermined set $S = \{s_i\}_{i=1}^{q}$. JK propose two mechanisms for checking the correctness of s_i. One is to generate the values and locations of blocks s_i during file encoding using a pseudorandom function (PRF), so that they are independent of F. The other appends a collection of q MACs to $\tilde{F}$ to allow checking of subsets of blocks of F. The JK protocol involves relatively small file expansion, as dictated by the error-correcting code, but supports only a limited number q of queries.

Ateniese et al. [1] propose a closely related construction called a *proof of data possession* (PDP). A PDP demonstrates to a client that a server possesses a file F (in an informal sense), but is weaker than a POR in that it does not guarantee that the client can retrieve the file. (In the nomenclature of proofs of knowledge, a PDP does not specify an *extractor*.) Curtmola et al. [7] describe how to integrate error-correcting codes with PDPs, and independently propose an adversarial error-correcting code construction similar to ours.

Shacham and Waters [21] propose protocols based on the idea of using *homomorphic authenticators* for file blocks, essentially block integrity values that can be efficiently aggregated to reduce bandwidth in a POR protocol. Due to the use of integrity values for file blocks, their scheme can use a more efficient erasure code to encode the file; the block authenticators transform the erasure code into an error-correcting code. Their scheme supports an unlimited number of verifications.

In concurrent and independent work, Dodis et al. [8] also give general frameworks for POR protocols that generalize both the JK and SW protocols. The focus of [8] is mostly theoretical in providing extraction guarantees for adversaries replying correctly to an arbitrary small fraction of challenges. In contrast, we consider POR protocols of practical interest (in which adversaries with high corruption rates are detected quickly) and show different parameter tradeoffs when designing POR protocols.

While the basic POR model supports checking of file retrievability by a single client in possession of a secret key, a *public* POR allows *any client* to verify the retrievability of F without secret keying material. Ateniese et al. [1] introduce the notion of public verifiability for PDPs. JK describe a straightforward Merkle-tree construction for public PORs, while SW describe a more efficient, public-key based version that relies on bilinear maps.

In other related work, Golle, Jarecki, and Miranov [12] propose techniques that enforce a minimum storage complexity on the server responsible for storing file F. They describe protocols that ensure dedicated use by a server of storage at least $|F|$ but do not enforce requirements on what data the server actually stores. Filho and Barreto [9] describe a POR scheme that relies on the knowledge-of-exponent assumption, first set forth in [3]. While communication efficient, this scheme is impractical, as respond requires computation of a modular exponentiation with respect to a bit-representation of all of F. Shah et al. [22] consider a symmetric-key variant of full-file processing to enable external audits of file possession. The scheme only works for encrypted files, and auditors are required to maintain long-term state.

Dynamic updates. File updates in POR/PDP protocols are a particular challenge. There is a strong tension between the security and efficiency of POR/PDP challenge-response protocols and the communication efficiency of file updates. Briefly, if an update changes a set A of blocks in the stored presentation of file F, then the server must "touch" A in computing a response to a challenge to ensure against irretrievable corruption of F. If A is a small, then "touching" A with high probability imposes high overhead on the server. In a basic POR, any change to the contents of file F, no matter how small, must propagate through the (generally substantial) error-correcting and challenge-response data encoded in $\tilde{F}$.

The lack of error correction and extraction algorithms in basic PDP constructions—and consequent lack of strong security against arbitrary adversaries—-permit file updates to be performed efficiently, i.e., with minimal communication overhead. In follow-up work to [1], Ateniese et al. [2] describe

---

[2]NR may be viewed as implicitly assuming that respond returns raw file blocks. It is possible, if awkward, to model nontrivial choices of respond in NR via an extended file encoding $\tilde{F}^* = \{\mathsf{respond}(\tilde{F}, c)\}_{c \in C}$ for challenge space C.

efficient tools for file updates in PDPs. Naor and Rothblum also support updates in their theoretical model.

We concern ourselves in this paper with archival and backup files. Such files are subject to infrequent change, so we treat them as static. Many systems modify backup or archival files periodically through incremental updates—often by appending a compendium of file changes. It is possible to treat such a compendium itself as an independent, static file in a POR system. That is, POR techniques may be applied to a full backup file by means of challenges against individual pieces: There is no need to reencode the full backup or archival repository.

That said, PORs do not gracefully handle files undergoing frequent, small updates. The specification of good batching schemes or other update techniques for PORs in such settings remains an open problem.

3. OUR CONCEPTUAL FRAMEWORK

3.1 Preliminaries

Following JK, a file $F = F_1, F_2, \ldots, F_m$ consists of a set of m blocks, each an l-bit symbol. We let L denote the symbol alphabet $\{0, 1\}^l$. (As a point of reference, in our implementation we work with 32-byte blocks, i.e., $l = 256$.) For brevity, we let P denote the prover (server or archive) and V denote the verifier (client). To draw on the formalization of JK, π denotes the set of system parameters, while ω denotes local, persistent client state. η is a file handle, which we drop from our notation where convenient. A POR includes six functions:

keygen$[\pi] \rightarrow \kappa$: Generates a secret key κ. (For a public POR, κ may be a public/private key pair.)

encode$(F; \kappa, \omega)[\pi] \rightarrow (\tilde{F}_\eta, \eta)$: Generates a file handle η and encodes F as a file $\tilde{F}_\eta$.

challenge$(\eta; \kappa, \omega)[\pi] \rightarrow c$: Generates a challenge value c for the file η.

respond$(c, \eta) \rightarrow r$: Generates a response r to a challenge c (the only function run by the prover).

verify$((c, r, \eta); \kappa, \omega) \rightarrow b \in \{0, 1\}$: Checks whether r is a valid response to challenge c. It outputs a '1' bit if verification succeeds, and '0' otherwise.

extract$(\eta; \kappa, \omega)[\pi] \rightarrow F$: Determines a sequence of challenges that V sends to P sufficient to recover the file and decodes the resulting responses. If successful, it outputs F.

3.2 Adversarial model

For our purposes in this paper, a challenge for a file F of size $|F|$ consists of a pair $c = (s, u)$, where $s \in [1, |F|]^v = S$ specifies a subset of v blocks in F, and $u \in [1, w] = W$ is an accompanying nonce. Here v and w are system parameters that specify the number of blocks included into a challenge, and the nonce domain size, respectively. We let C denote the full challenge space $S \times W$.

An adversary A in our model receives an encoding of the file, and needs to answer at most N_{max} challenges over the lifetime of the file. The adversary is determined by a set of partitions $(C_i^+, C_i^-)_{i=1, N_{max}}$ of C. C_i^+ is the set of all challenge values (s, u) to which the adversary responds correctly when queried for the i-th time, while C_i^- is the set to which it responds incorrectly at the i-th challenge. The partition (C_i^+, C_i^-) is adaptively determined for every $i = 1, N_{max}$.

We let $\epsilon_{s,u}^i = 1$ if $(s, u) \in C_i^-$, and $\epsilon_{s,u}^i = 0$ otherwise. We let $\epsilon_s^i = \sum_{u \in W} \epsilon_{s,u}^i / w$ be the probability that the adversary responds incorrectly the i-th time to challenges on a subset s of blocks for a random choice of u. We let $\epsilon^i = \frac{\sum_{s \in S, u \in W} \epsilon_{s,u}^i}{|C|}$ be the probability that the adversary responds incorrectly the i-th time to a challenge selected uniformly at random from C, i.e., $\epsilon^i = |C_i^-|/|C|$. We denote $\epsilon^{\mathcal{A}} = \max_{i=1, N_{max}} \epsilon^i$ the maximum fraction of corrupted challenges at each query.

An ϵ-adversary is one for which $\epsilon^{\mathcal{A}} \leq \epsilon$. Intuitively, an ϵ-adversary replies correctly to a fraction of at least $1 - \epsilon^{\mathcal{A}}$ challenges over the lifetime of the file.

3.3 Our POR framework: Key ideas

It is useful to think of a POR in our framework as a two-phase process:

3.3.1 Phase I: Ensuring an ϵ-adversary

In the first phase of a POR (depicted in Figure 1), the client performs a series of challenge-response interactions with the server $\mathcal{A}$ over file $\tilde{F}_{out}$ (i.e., the encoding of F under the outer code ECC_{out}), with the aim of detecting the condition $\epsilon^{\mathcal{A}} > \epsilon$. To challenge the server, the client computes $c = \mathsf{challenge}(\eta; \kappa, \omega)[\pi]$, sends c to the server, receives a response r, and then computes $\mathsf{verify}((c, r, \eta); \kappa, \omega)$ to check the response of the adversary. The client repeats this process q_c times, and rejects if any response is incorrect. Otherwise the client accepts.
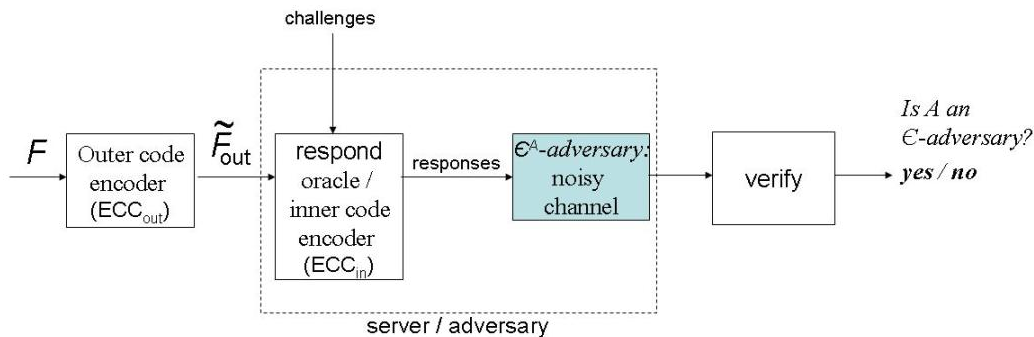
Assuming that challenge selects $c \in_U C$, the probability that an adversary $\mathcal{A}$ is accepted but is *not* an ϵ-adversary, i.e., $\epsilon^{\mathcal{A}} > \epsilon$, is $< \lambda = (1 - \epsilon)^{q_c}$. The value λ can be made arbitrarily small, with an appropriately large q_c.

The JK protocol checks adversarial responses by precomputing a challenge set $\{c_i\}_{i=1}^q \in_U C$ and storing verifying data—sentinels or MACs—in the encoded file for download by the client. The Lillibridge et al. and NR constructions check adversarial responses by verifying MACs on file blocks. Thus, both of these constructions also select $c \in_U C$. The SW scheme omits Phase I, i.e., implicitly assumes an ϵ-adversary.

Remark. In practice, as envisioned in our implementation, the server may initially be honest, but turn bad at some point and be replaced by an adversary $\mathcal{A}$. To deal with such a dynamic adversary, the client may spread out its challenges over time. For example, the client might initiate a challenge every day. If Phase I is tuned to achieve a particular λ for $q_c = 50$, then, the condition $\epsilon^{\mathcal{A}} > \epsilon$ will be detected with probability at least $1 - \lambda$ within the first 50 days after the server has turned adversarial.

3.3.2 Phase II: Extracting F from an ϵ-adversary

Assuming an honest server, a client can simply download F—and verify its correctness via an appended MAC or digital signature. If this fails, the client can download the encoded file $\tilde{F}_\eta$ and try to correct it using the outer error-correcting layer. Failing that, given an ϵ-adversary, it is possible for the client to retrieve F via extract, executing a series of challenges and decoding F from the responses. Note that in this phase, the client may not be able to verify the correctness of the responses it receives (in particular, for protocols with a bounded number of verifications): In this case, it relies on the ϵ-bound on $\mathcal{A}$ for

Figure 1: Schematic of Phase I in our POR framework: Testing whether $\epsilon^{\mathcal{A}} \leq \epsilon$, i.e., if $\mathcal{A}$ is an ϵ-adversary.

successful decoding.

In our general framework, there are *two levels of error-correction*:

- The *outer code*: This is a (n, k, d_2)-error-correcting code ECC_{out} applied to F to compute $\tilde{F}_{out}$, the error-corrected portion of F output by encode. Usually, for large files, $n < m$, and to encode a file of size m we need to resort to a well-known technique, called *striping*: the file is divided into stripes of size k blocks each, and each stripe is encoded under ECC_{out}. In the rest of the paper, when we encode the file with the outer code, we implicitly mean that striping is performed if necessary.

- The *inner code*: This (w, v, d_1)-error-correcting code ECC_{in} represents a second layer of error-correction in the challenge-response interface for a POR. The function $\mathsf{respond}(s, u)$ applies ECC_{in} to the set s of message blocks specified in a challenge; the value $u \in W$ specifies which symbol of the corresponding codeword should be returned to the client.

In this view, the adversary is a noisy channel with error probability at most ϵ. We may think of the adversary as intercepting transmissions from a (correct) oracle for respond to the client. When the client submits the i-th challenge $c = (s, u)$, the respond oracle computes the correct response r. If $\epsilon^i_{s,u} = 1$, then $\mathcal{A}$ corrupts the response in the channel; otherwise, the adversary leaves r unchanged.

The goal of employing two levels of error correction in the design of our POR framework is to correct the adversarial error ϵ. The effect of the inner code is to drive down the adversarial error ϵ to some error value $\epsilon' < \epsilon$. The outer code then corrects this residual error ϵ'. Thus, the stronger the inner code, the weaker the outer code we need to employ. The outer code needs to be an *adversarial error-correcting code* (defined in the full version of the paper [6]). Intuitively, an adversarial code transforms an ϵ' computationally bounded adversary to a random one, i.e., one in which the adversary has no better chance of corrupting codeword symbols than choosing at random.

Our PORs are designed to be effective against a maximum adversarial error ϵ. In the next section, we will show examples of protocols that tolerate ϵ values non-negligibly close to 1, as well as protocols that require an upper bound on ϵ less than $\frac{1}{2}$. There are two main types of POR protocols that we can construct in our framework:

1. In protocols that enable an unbounded number of server verifications (e.g., SW), the responses to challenges in Phase II can be verified. In this case, the inner code can simply be an erasure code and can therefore tolerate error rates ϵ non-negligibly close to 1. We describe in Section 4 a generalization of SW that fits into our framework. Even if, theoretically, such protocols do not need to employ Phase I to limit ϵ (since they can extract for ϵ close to 1), in practical settings obtaining a Phase-I bound on the adversarial corruption rate is still valuable to ensure efficient extraction.

2. In protocols in which a limited number of responses can be verified, we need to employ an error-correcting inner code in order to correct arbitrary server responses. In such protocols, we need to set an upper bound on the adversarial rate less than $\frac{1}{2}$ and dependent on both the inner and outer code parameters. For such protocols to fit our framework, we need to assume that the adversary's corruption rate during extraction is also limited by the same $\epsilon < \frac{1}{2}$. Under this assumption, either the adversary corruption rate is greater than ϵ, and will be detected with high probability in Phase I, or ϵ is low enough so that we can extract F successfully in Phase II.

   In Section 4 we present a more efficient version of JK that employs a limited number of verifications assuming an upper bound on ϵ given by the error correction rate of the inner code. Despite such restrictions, we show its practical advantages in storage overhead and proof costs compared to SW for values of ϵ within the error-correction capabilities of the inner and outer codes.

Now the full storage and successful extraction process for the client is as follows. We let the superscript $*$ denote a corrupted file:

1. **Outer encoding and storage:** The client encodes file F under ECC_{out} as $\tilde{F}_{out}$, and stores $\tilde{F}_{out}$ with the server. (Thus $\tilde{F}_{out}$ is a component of the full file encoding $\tilde{F}_\eta$, which may include supplementary data such as MACs.)
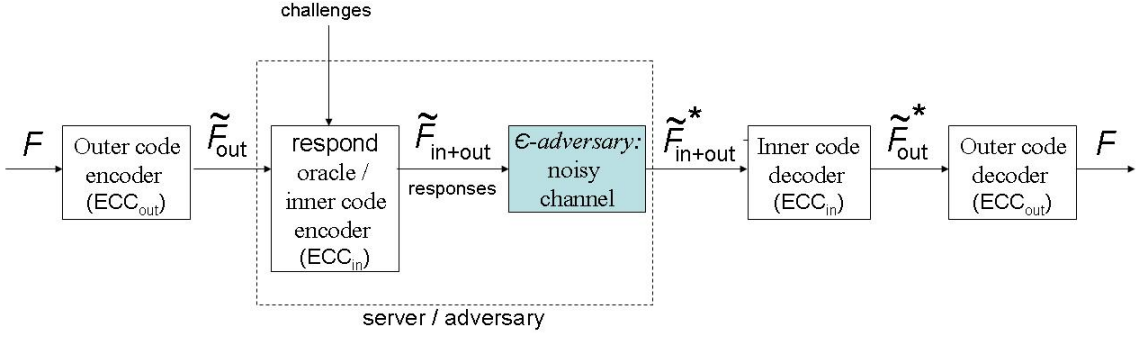
5

Figure 2: Schematic of Phase II in our POR framework: Extracting F from an ϵ-adversary.

2. **Extraction:** If both ordinary downloading of F and error-correction of $\tilde{F}_\eta$ fails, the client invokes extract. In this case, the client submits a series of challenges to the respond oracle, which outputs symbols under the encoding ECC_{in}. Together, these responses make up a file $\tilde{F}_{in+out}$ that is encoded under a composition of ECC_{out} and ECC_{in}.

3. **Corruption / noise:** The adversary corrupts up to an ϵ-fraction on average of $\tilde{F}_{in+out}$. The resulting file is $\tilde{F}^*_{in+out}$.

4. **Inner decoding:** The client decodes the inner code in $\tilde{F}^*_{in+out}$ under ECC_{in}, obtaining file $\tilde{F}^*_{out}$. The file $\tilde{F}^*_{out}$ is a representation of $\tilde{F}_{out}$ with ϵ'-fraction corruption, where $\epsilon' < \epsilon$.

5. **Outer decoding:** The client decodes $\tilde{F}^*_{out}$ under ECC_{out}, obtaining the original file F.

3.4 Security definition

By viewing a POR as a two-phase process, we are able to offer a simpler security definition than JK that is akin to that of SW. We abstract away Phase I, and assume an ϵ-adversary. Referring then to JK for details of the experimental setup:

DEFINITION 1. *A poly-time POR system* $\mathsf{PORSYS}[\pi]$ *is a* (ϵ, γ)-*valid proof of retrievability (POR) if for every poly-time* ϵ-*adversary* $\mathcal{A}$*, the probability that* extract *outputs* F *is* γ. *More formally,*

$$\gamma = \mathsf{pr}\big[F = F_{\eta^*} \mid F \leftarrow \mathsf{extract}^{\mathcal{A}(\delta, \cdot)(\text{``respond''})}(\eta^*; \kappa, \omega)[\pi]\big].$$

Remark. Observe that the inner code imposes no storage overhead in our protocol, as it is computed on the fly by respond. We could in principle use an inner code only, with no outer code. The drawback to this approach is that the message size of ECC_{in} would have to be very large to guarantee extraction. In the limit, we could let $v = m$. In this case, respond would treat the whole file F as a message, and return one symbol of a codeword over the whole file. In this case, if $d_1 \geq 2\epsilon m$, i.e., the code ECC_{in} is resilient to an ϵ-fraction of file corruption, we could dispense with the outer code. But in practice, constructing an error-correcting code with large message size and large distance would be impractical. So in constructing a POR, we seek to strike a balance between the resource requirements of the inner code (computation and file-block retrieval) with the outer code (file expansion).

4. NEW VARIANT POR PROTOCOLS

4.1 The SW scheme

SW propose two constructions based on the idea of storing *homomorphic authenticators* for file blocks together with the file. These are essentially block integrity values that can be efficiently aggregated to reduce bandwidth in a POR protocol. Since each block can be independently verified with the stored authenticators, SW could encode the file with a more efficient erasure code. The block authenticators (or MAC values) effectively turn the erasure code into an error-correcting code.

More specifically, to encode a file F of length $|F|$ in the SW scheme, an erasure code of rate ρ is applied first. Then the encoded file is divided into n blocks, each consisting of s sectors. A sector is an element of a prime group Z_p, where the size of p is a security parameter λ. Thus, $|F| = ns\lambda\rho$.

For each block $1 \leq i \leq n$, an authenticator σ_i is computed and stored with the file. In addition, the client stores a file tag of size $s\lambda$. In a challenge, the client in the SW scheme sends l block indices and l values in Z_p. The server response is one aggregated authenticator of size $(s + 1)\lambda$.

The SW construction fits nicely into our framework. SW implicitly assumes an ϵ-adversary (for some $\epsilon < 1$) and thus omits Phase I of our framework. The SW scheme setup, however, allows us easily to include a Phase I stage to bound ϵ below any desired value. While SW can tolerate essentially any $\epsilon < 1$, the ability to bound ϵ at lower values in a Phase I stage can be quite important in practice. The error value ϵ determines the efficiency of file extraction for the client, so ensuring ϵ bounded well away from 1 can be vital in practice.

We can generalize the SW construction by replacing their ad-hoc inner erasure code with an erasure code with linear encoding and decoding time, e.g., Tornado codes [16], on-line codes [17], LT codes [15] or Raptor codes [23]. In addition to obtaining a more efficient extraction algorithm (reducing the decoding cost from $O(|F|\sqrt{|F|})$ to $O(|F|)$, for the natural choice of $n = s = \sqrt{|F|}$), this approach can simplify the security proofs of SW. For instance, the proofs from Section 4.2 in [21] could be easily inferred from the decoding properties of the above mentioned erasure codes.

4.2 Improvement to JK protocol

We now describe improvements that our new framework

brings to the JK protocol, the main focus of the paper.

As explained above, SW has strong theoretical benefits, e.g., an ability to extract files for essentially any $\epsilon < 1$. In practice, though, an adversary with $\epsilon > \frac{1}{2}$ would be quickly detected in phase one of our framework. Consequently, in practice, the higher ϵ tolerated by SW is of limited benefit. SW also permits an unbounded number of phase-one queries by the client. JK, however, can support a large number of phase-one queries in practice. As we explain in Section 5, JK has notable efficiency benefits over SW for parameter ranges we consider of most practical interest. This is the reason for our focus on JK.

Our main goals in designing the new variant on JK are to tolerate a larger level of errors than in the original JK scheme, reduce the storage overhead on the server, and employ a more lightweight verification mechanism in the first phase to ensure that the client is dealing with an ϵ-adversary. After describing the details of the new variant, we provide its security analysis in a stronger adversarial model than employed in the JK protocol. We conclude the section by showing a range of parameters our new variant supports and their relative tradeoffs.

Building blocks

To specify the algorithms in our protocol, we need several cryptographic primitives, in particular a symmetric-key authenticated encryption scheme ($\mathsf{KGenEnc}, \mathsf{Enc}, \mathsf{Dec}$), a family of pseudorandom permutations $\mathsf{PRP}[n] : \mathcal{K}_{\mathsf{PRP}} \times \{0,1\}^n \to \{0,1\}^n$ and a message-authentication algorithm ($\mathsf{KGenMAC}, \mathsf{MAC}, \mathsf{Ver}$). We assume the primitives are secure according to standard security definitions, i.e., the encryption scheme is IND-CPA secure, the permutation family is pseudorandom and the MAC scheme is unforgeable [10, 11].

Outer layer of error correction

The outer layer of error correction needs to be adversarial in the sense that it transforms a noisy channel into a random one (see a discussion on adversarial codes in the full version of the paper [6]). To implement an adversarial code, the JK's encode algorithm uses a striped "scrambled" code. The file is divided into stripes and each stripe is encoded with a standard (n, k, d) Reed-Solomon code. The resulting symbols are permuted with a PRP and encrypted with secret keys known only to the client.

Our goal is to construct a systematic adversarial error-correcting code, i.e., one in which the message blocks of F remain unchanged by error-correcting. A systematic code of this kind has considerable practical benefit. In the ordinary case when the server is honest and extraction is unnecessary, i.e., the vast majority of the time, the client need not perform any permutation or decryption on the recovered file. To build a systematic adversarial error-correcting code, we apply code "scrambling" *exclusively to parity blocks*. However, this does not ensure by itself a random adversarial channel, since the adversary has information about stripe boundaries. To hide stripe boundaries from the server, we first implicitly apply a pseudorandom permutation to the file blocks and then divide the permuted file into stripes. Our outer code outputs the file in order, followed by the "scrambled" parity blocks.

The encode algorithm of our adversarial code SA-ECC takes as input secret keys k_1, k_2 and k_3, and a message M of size m blocks, and performs the following operations:

- Permute M using $\mathsf{PRP}[m]$ with key k_1, divide the per-

muted message into $\lceil \frac{m}{k} \rceil$ stripes of consecutive k blocks each, and compute error-correcting information for each stripe using code ECC_{out}.
- The output codeword is M followed by permuted and encrypted error-correcting information (the permutation of parity blocks is done using $\mathsf{PRP}[\frac{m}{k}(n-k)]$ with secret key k_2 and their encryption with key k_3, respectively).

Remark.

We emphasize that thanks to our use of systematic encoding of F, we can achieve strong security in the encoded file by *encrypting only the parity blocks*. Such limited encryption fully conceals the error-coding stripes, as the blocks of F itself remain untouched and therefore carry no permutation information.

The decode algorithm of SA-ECC simply reverses the operations of encode.

Precomputing challenge-response pairs

As described in Phase II of our framework, a challenge is a pair $(s, u) \in S \times W$. The response is the u-th symbol in a codeword over an inner coding instance consisting of file blocks with indices in set s. The client precomputes a set of randomly generated challenges derived from a seed and appends to the file encrypted (and authenticated) corresponding responses.

4.2.1 Complete POR protocol

We present here the complete POR protocol. In the keygen algorithm, a master secret key MS is generated, from which additional keys are derived: a master challenge key k_{chal}, a key k_{ind} used to sample codeword indices of ECC_{in}, a file MAC key k_{MAC}^{file}, a master encryption key k_{enc}, a file permutation key k_{perm}^{file}, an ECC permutation key k_{perm}^{ECC} and an ECC encryption key k_{enc}^{ECC}. Let us denote the generator matrix of ECC_{in} by $G = \{g_{ij}\}_{1 \le i \le v, 1 \le j \le w}$.

The encode algorithm for our POR protocol is the following:

1. Divide file F into m blocks, each an l-bit symbol, i.e., $F = F_1 \dots F_m$.
2. Apply outer error-correcting layer: Encode F under SA-ECC with secret keys ($k_{perm}^{file}, k_{perm}^{ECC}, k_{enc}^{ECC}$), resulting in $F' = F_1 \dots F_m F_{m+1} \dots F_t$ (with $t = \lceil \frac{mn}{k} \rceil$).
3. Precompute challenge-response pairs: For each challenge j with $1 \le j \le q$:
    (a) The client first derives a challenge key k_j^c from k_{chal} from which she computes v pseudorandom block indices $i_1, \dots, i_v \in [1, t]$. The client derives a random index $u \in [1, w]$ from seed k_{ind} and an encryption key k_j^e from k_{enc}.
    (b) The client computes $M_j = \sum_{s=1}^{v} F_{i_s} g_{su}$ and appends $Q_j = \mathsf{Enc}_{k_j^e}(M_j)$ to the encoded file.
4. Append a MAC of the whole file $\mathsf{MAC}_{k_{MAC}^{file}}(F)$ and return the encoded file.

In the challenge algorithm, the client sends to the server j, k_j^c and the random index u derived from k_{ind}. In the respond algorithm, the server derives $i_1, \dots, i_v$ from k_j^c, computes

$M_j = \sum_{s=1}^{v} F_{i_s} g_{su}$, and returns to the client M_j and Q_j. The verify algorithm returns true if $M_j = \mathsf{Dec}_{k_j^e}(Q_j)$.

In the extract algorithm, executed when normal file download and error-correction of the encoded file fail, the client executes two phases, one for each layer of error correction. To decode from the inner layer, the client submits a sufficient number of challenges and then uses majority decoding. The client obtains on average α decodings for each file block (for α a parameter in the system chosen so that each file block is covered with sufficiently large probability), and she decodes to the block that appears in at least a fraction of $\frac{1}{2} + \delta$ of decodings (for $\delta > 0$ a parameter of our system). If no such block exist, the client outputs an erasure for that block, denoted $\perp$. For each file block i, the client maintains during the first extraction phase a set of all decodings obtained with repetitions, denoted $\mathcal{D}_i$. After decoding all blocks in the first phase, the client uses the decoding procedure of the outer error-correcting code in order to correct the possible errors and erasures introduced in the first phase. The extract algorithm is given below:

1. Recover from the inner error-correction layer

    (a) $\mathcal{D}_i = \Phi$, for all blocks $i \in [1, t]$.
    (b) Pick a set of challenges $\mathcal{C}$ of size $N_{\mathcal{C}} = \alpha \frac{t}{v}$ as follows:

        (b1) For each $j \in [1, \alpha \frac{t}{v}]$ do:
            • Generate a seed k_j^c (used to generate a sequence of v block indices).
            • Add (j, k_j^c) to $\mathcal{C}$.

    (c) For each challenge $(j, k_j^c) \in \mathcal{C}$ do:

        (c1) Execute challenge w times with parameters j, k_j^c and u, where u takes all the values between 1 and w and all the other parameters remain constant.
        (c2) Apply the decoding procedure of ECC_{in} to recover $F_{i_1}, \ldots, F_{i_v}$ (where $i_1, \ldots, i_v$ are generated from seed k_j^c) and add each F_{i_s} to the set $\mathcal{D}_{i_s}$, for $s \in [1, v]$.

    (d) For each block $i \in [1, t]$ do:

        (d1) If there exists $b \in \mathcal{D}_i$ such that $\frac{|j : \mathcal{D}_i[j] = b|}{|\mathcal{D}_i|} \geq \frac{1}{2} + \delta$, output $F_i = b$.
        (d2) Otherwise, output $F_i = \perp$.

2. Recover from the outer error-correcting layer: Decode $F_1 \ldots F_t$ under SA-ECC using secret keys ($k_{\mathsf{perm}}^{\mathsf{file}}$, $k_{\mathsf{perm}}^{\mathsf{ECC}}$, $k_{\mathsf{enc}}^{\mathsf{ECC}}$) and obtain F.

3. Compute the MAC over the whole file and check it against the MAC stored at the end of the file. If the MAC verifies, output the file, and otherwise output error.

The full security analysis of this construction is given in the full version of the paper [6].

4.2.2 Parameterization

In this section, we show different tradeoffs our construction achieves for different parameter choices. We consider different types of inner codes, in particular a first class of theoretical codes whose existence is guaranteed by the Varsharmov-Gilbert lower bound, and a second class of practical codes that can be easily built from concatenation of standard Reed-Solomon codes.

Theoretical codes.

The following lower bound for the minimum distance of a code (n, k) holds.

THEOREM 1. *(Varsharmov-Gilbert [20]) It is possible to construct an (n, k) code over an alphabet Σ of size σ with minimum distance at least d, provided that:* $\sum_{i=0}^{d-2} \binom{n}{i} (\sigma - 1)^i \geq \sigma^{n-k}$.

We consider several codes over a byte alphabet that follow the lower bound by varying codeword sizes from 500 to 4000 and code rates from 0.1 to 0.9.

Practical codes obtained from concatenation.

A standard code used in practical applications is the systematic $(255, 223, 32)$ Reed-Solomon code. From this code we build several codes $(k + 32, k, 32)$, with $32 \leq k \leq 223$, by padding k with zeros to obtain a message of size 223, encoding the padded message, and truncating the codeword of size 255 to size $k + 32$. It is easy to see that the distance given by this code remains 32. We obtain by this procedure codes $(64, 32, 32)$ and $(96, 64, 32)$.

For our construction, we need inner codes that operate on larger message sizes, on the order of several thousand bytes. A standard procedure to enlarge the message and codeword sizes is to build concatenated codes. The concatenation of two codes with parameters (n_1, k_1, d_1) and (n_2, k_2, d_2) is denoted $(n_1, k_1, d_1) \cdot (n_2, k_2, d_2)$ and has parameters $(n_1 n_2, k_1 k_2, d_1 d_2)$. A description of the concatenation procedure is outside the scope of this paper, but we refer the reader for more details to [20]. Several codes obtained through concatenation are given in Table 1.

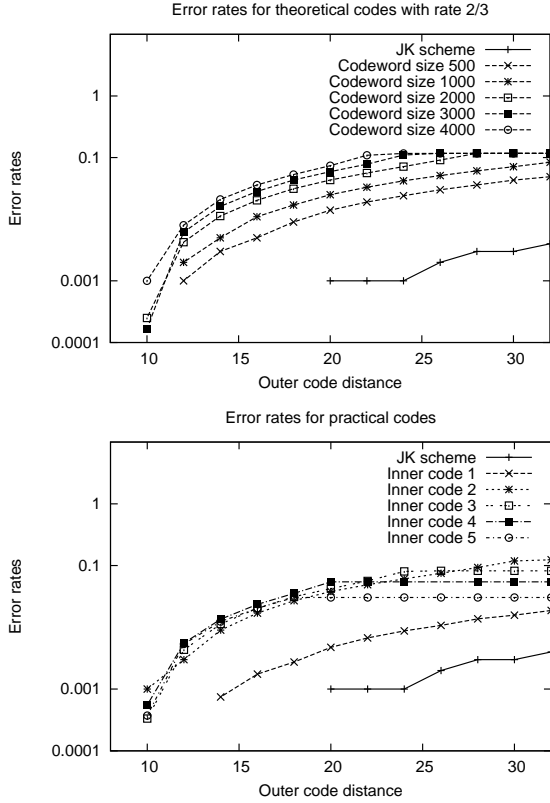| Code name	Code parameters	How obtained
Code 1	(255,223,32)	Reed-Solomon code
Code 2	(4096,1024,1024)	$(64, 32, 32) \cdot (64, 32, 32)$
Code 3	(6144,2048,1024)	$(64, 32, 32) \cdot (96, 64, 32)$
Code 4	(9216,4096,1024)	$(96, 64, 32) \cdot (96, 64, 32)$
Code 5	(16320,7136,1024)	$(64, 32, 32) \cdot (255, 223, 32)$

Table 1: Several practical codes.

Error rates tolerated.

In our new variant, we can tolerate a higher error rate than in the JK scheme, due to the addition of a new dimension in the design space of POR protocols, namely the inner code. As explained in our theoretical framework, the inner code reduces the adversarial error ϵ to a residual $\epsilon' < \epsilon$, and the outer code then corrects the residual error ϵ'. Since in the new variant ϵ' is reduced by at least an order of magnitude compared to ϵ, we need a more lightweight outer code than the JK scheme, and, as such, the outer code storage overhead decreases.

We show in Figure 3 the maximum error rate ϵ tolerated by different inner codes that follow the Varsharmov-Gilbert lower bound, as well as by several practical codes. For theoretical codes, we fix the code rate to a constant (2/3, corresponding to an expansion of 50%), and only vary the codeword size (and, implicitly, the message size)[3]. The graphs from Figure 3 plot

---

[3]We also performed tests for theoretical codes with rates varying from 0.1 to 0.9 with a 0.1 increment. It turns out that similar results hold, and, thus, we omit them from the paper.

Error rates for theoretical codes with rate 2/3

Number of challenges for theoretical codes with rate 2/3

Error rates for practical codes
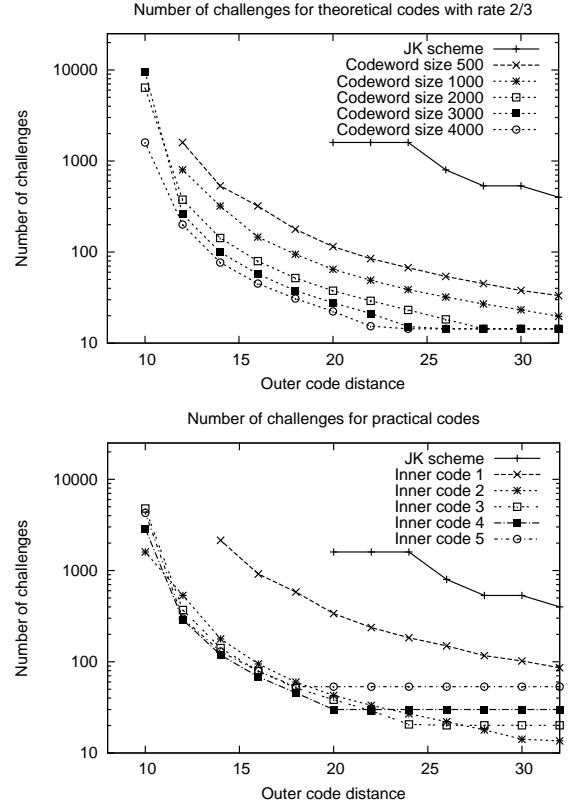

Number of challenges for practical codes

Figure 3: Maximum error rate ϵ tolerated as a function of outer code distances for both theoretical inner codes with rate 2/3 and for different practical codes.

Figure 4: Number of challenges required in Phase I as a function of outer code distance.

the tolerated error rates as a function of the outer code distance, for a file size of 4GB, $\alpha = 10$, $\delta = \frac{1}{4}$ and security bound $\gamma = 10^{-6}$. Our outer code is built from the standard $(255, 223, 32)$ Reed-Solomon code, by truncating codewords to size $223 + d$ to obtain distance $0 < d \leq 32$.

When the outer code distance drops below a certain threshold (i.e., 20 for the JK scheme, and between 10 and 14 for the new variant), we can no longer obtain a security bound of $\gamma = 1 - 10^{-6}$. This shows that our new variant spans a larger parameter domain, allowing different tradeoffs between the outer code storage overhead, the error rates tolerated and the number of verifications in Phase I.

The results for theoretical codes show that for a fixed outer code distance (and, implicitly, outer code storage overhead), higher error rates are tolerated by codes with larger codeword sizes. For practical codes, the results demonstrate that error rates do not depend only on inner codeword size, but also on inner code rate and minimum distance. For instance, for an outer code distance greater than 26, inner code 2 tolerates a higher fraction of errors than inner codes 3-5, even though its codeword size is smaller. For outer code distances smaller than 22, the amount of errors tolerated by codes 2-5 is close, with a difference of at most 0.003 between any consecutive codes. Code 1 performs much worse than codes 2-5 due to its small codeword size of 255 bytes.

Number of challenges required in Phase I.

Intuitively, as the protocol tolerates a larger amount of error rates, the number of verifications needed in Phase I of our framework decreases. Figure 4 shows the number of challenges for both theoretical codes with rate 2/3 and practical codes. For an outer code distance of 32 (and, thus, a file storage overhead of 14.34%), inner code 3 requires only 20 challenge verifications in Phase I. In contrast, JK needs 400 verifications for the same security level. The number of verifications in Phase I becomes prohibitive very fast for JK: for an outer code distance of 24, 1596 verifications are necessary. In contrast, as the outer code storage overhead decreases from 32 to 16 in the new variant, the number of challenges increases at an almost linear rate. For instance, for an outer code distance of 16, we need to check 94 challenges with inner code 2, 79 with inner code 3, and 68 with inner code 4. When the outer code distance drops below 16 in the new variant, the number of challenges exhibits an exponential increase. In conclusion, using one of the inner codes 2, 3 or 4 in the new variant, we can obtain a 50% reduction in the outer code storage overhead, at the expense of checking less than 100 challenges in the first phase.

5. PRACTICAL COMPARISON WITH PREVIOUS SCHEMES

Scheme	SW	Our POR
Storage overhead on server	$n\lambda + (\frac{1}{\rho} - 1)\|F\|$	$zq + (\frac{1}{\rho_{out}} - 1)\|F\|$
Storage cost on client	$s\lambda + \kappa$	κ
Size of challenge	$l\lambda$	$8 + \kappa$
Size of response	$(s+1)\lambda$	z
Parameters	$\|F\| = ns\lambda\rho, \rho^l = 2^{-\lambda}$	$\kappa = 32$ bytes, $z = 32$ bytes

Table 2: Storage overhead on server and client, and communication cost for SW and our scheme.

5.1 New Variant JK Protocol vs. Original JK Scheme

The original JK scheme employs only one layer of error-correction. In the challenge phase, the client downloads a number of sentinel values from the server and verifies their correctness. Thus, the challenge phase in JK is only used to ensure an ϵ-adversary, but is not effectively useful to extract file blocks. For this reason, their scheme is resilient to a smaller fraction of block corruptions than the new variant. Our extraction success is amplified by using two layers of error correction.

We performed a detailed comparison of the two schemes by using the results from Figures 3 and 4. The new variant tolerates an error rate ϵ at least an order of magnitude higher than the original JK scheme for the same outer code overhead and security bound. This imposes in the JK scheme the verification of a larger number of challenges in the first phase (e.g., by a factor between 20 and 80 for inner code 3) to achieve an equivalent error rate. The cost we pay for our efficiency in storage overhead and first phase verification is a more expensive extract algorithm. However, our hope is that in the normal case, i.e., most of the time, the user downloads the original file with a valid MAC or can correct the encoded file using ECC_{out}, and does not need to resort to extract for file recovery.

Moreover, our security analysis for the new variant is performed in a much stronger adversarial model, since we do not make simplifying assumptions about the adversary's behavior, except for the bound on ϵ as described above. JK includes a strong *block isolation assumption*, stating that the probabilities file blocks are returned correctly in a challenge are independent of one another.

5.2 Comparison to Shacham-Waters Scheme

5.2.1 Parameters in Shacham-Waters

In Table 2, we quantify the server storage overhead cost, storage cost on the client, and communication cost of challenge and response protocols in both SW and our new POR scheme. SW is parameterized by security parameter λ, the rate of the erasure code ρ, the number of blocks and authenticators n, and the size of a block s, such that ρ^l is negligible in λ, e.g., $\rho^l = 2^{-\lambda}$. The JK variant is parameterized by $\rho_{out} = \frac{k}{n}$ the rate of the outer code ECC_{out}, z the size of a symbol in the outer code (e.g., 32 bytes in our implementation), and κ the size of symmetric keys (e.g., 32 bytes).

5.2.2 Discussion

While we have crafted our scheme to trade off the stor-age overhead on the server against the number of challenges needed to ensure an ϵ-adversary in the first phase, and the extraction complexity, we observe a different tradeoff in the design of SW: The storage overhead on the server, storage on the client, and communication complexity in challenge and response protocols are traded off by the choice of parameters n, s and the erasure code expansion rate ρ.

Let us consider a particular parameterization of our scheme in which we set a security level of 10^{-6}, an outer code with rate 0.9 (and thus an expansion of 10%), and the total number of pre-computed challenges stored $q = 10000$. The storage overhead on the server for our scheme is 320KB + 0.1 · $\|F\|$. The maximum adversarial corruption rate $\epsilon < \frac{1}{2}$ is a value bounded by our inner and outer error correcting codes (for instance, for the $(4096, 1024, 1024)$ inner code and the $(255, 223, 32)$ outer code, ϵ could be as large as 0.1 as shown in Figure 3). We show how we can design SW to match some of the parameters, assuming the same bound on ϵ. For SW, a security level of 10^{-6} translates to $\lambda = 22$.

To illustrate the relative costs between SW and our scheme, we consider two points of comparison, one in which the server storage overhead is set to be identical in both schemes, and another in which communication complexity is set to be identical. We derive the parameters for a 4GB file.

- For the same storage overhead on the server for the two schemes, we need to set ρ to 0.1 in SW, and thus $l = 169$. Additionally, $n\lambda$ needs to equal 320KB, and we can derive $s = 13,107$. In this case, the size of a challenge in SW is 464 bytes, and the response size is 36KB, as opposed to both these values being on the order of 40 bytes in our construction. The communication complexity in SW is increasing with the file size.

- For the same communication complexity in the two schemes, $l\lambda$ needs to be 40 bytes, and $(s+1)\lambda$ needs to be 32 bytes, which implies $l = 14$ and $s = 10$. We can then derive $\frac{1}{\rho} = 2.96$, and thus the storage overhead on the server in SW is about twice the size of the file.

These examples show that for small values of ϵ SW is more expensive than our scheme with respect to both storage overhead on server and client and communication complexity. But, of course, it still retains the advantage that it allows for an unlimited number of verifications, and can provide file extraction for any $\epsilon < 1$ (albeit at polynomial cost in $\frac{1}{1-\epsilon}$).

6. IMPLEMENTATION

There are several challenges that we have encountered in the process of implementing the new variant.

6.1 Small PRPs

To build adversarial code SA-ECC, we need to construct two pseudorandom permutations: one that permutes file blocks to generate stripes, and the second that permutes parity blocks. Both are "small" pseudorandom permutations, i.e., smaller than the size of a typical block cipher. For instance, for a 4GB file divided into 32-byte blocks, we need a permutation with domain size of 2^{27} to permute file blocks.

Black and Rogaway [4] have considered the problem of designing small block ciphers, and proposed several solutions.

For small domains, a practical solution (method 1 in [4]) is to build and store into main memory a table with random values. Another method applicable to larger domains (method 3 of [4]) is to use a 3-round Feistel cipher, with the random functions in each round based on a standard block cipher, such as DES or AES. However, their security bound is quite weak, i.e., the PRP-advantage to generate a permutation of length $2n$ is on the order $\frac{q^2}{2^n}$, where q is the number of permutation queries asked by the adversary.

We have found a solution to enhance this security bound in a paper by Patarin [19]. Patarin proves that, if 6 rounds are performed in the Feistel construction, then the indistinguishability advantage of a permutation of size $2n$ is $\frac{5q^3}{2^{2n}}$. This bound is ideal for our limited adversarial model, in which $q = 1$ (our adversary does not have access to a PRP oracle). The method we adopted in our implementation is to use a 6-round Feistel construction, with the random functions in each round implemented as random tables stored into memory. We have briefly experimented with the random functions implemented with AES, but this alternative is several orders of magnitude slower than having the tables stored into main memory.

6.2 Incremental encoding

A naive implementation of our encode protocol based on random access to file blocks for encoding would be prohibitively expensive (for a seek time of 10ms, it takes 66000s to randomly access a 1GB file, using 32-byte blocks). As solid-state disks become more popular, random access might become comparable to sequential access. In the meanwhile, for today's drives, our objective is to implement the encoding algorithm in only *one pass* through the file, in an incremental fashion.

If Reed-Solomon codes are used for implementing ECC_{out}, computation of parity blocks involves associative operations in a Galois field. For an incremental encoding of the file, we could store the parity blocks into main memory, and update them when we process a file chunk in memory. However, we pay some performance cost for incremental encoding, since existing Reed-Solomon implementations are highly optimized if the whole message is available at encoding time. The parity block size depends on both the file size and the outer code distance, but for a fixed outer code distance, it grows linearly with the file size.

With this method, we could encode incrementally files for which the parity blocks (whose size is a percentage of the file size) fit into memory. For large files for which parity blocks do not fit into main memory, there are several encoding alternatives. First, we could still perform a single pass over the file, but store only a smaller set of parity blocks into memory. As a file block is processed in memory, the parity blocks that depend on it are swapped in, updated, and then swaped out of memory. Another alternative is to break a very large file into more manageable sized files, and encode each separately.

Another problem we encountered was how to prepare the challenges in an incremental fashion. To compute each challenge, v blocks need to be picked at random from the range of all file blocks. A simple solution, which we have implemented, is to generate in advance all these v block indices for each of the q challenges, and store them into main memory. We have experimented with an alternative technique in which
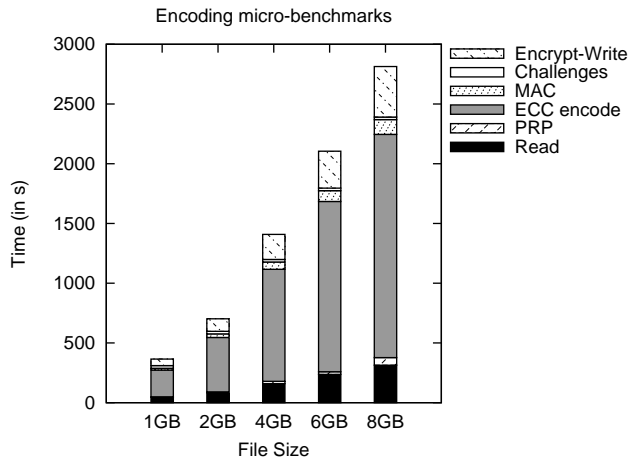

Figure 5: Encoding micro-benchmarks for inner code $(4096, 1024, 1024)$ **and outer code** $(241, 223, 18)$**.**

we select the number of samples for each chunk from a Binomial distribution, and then sample uniformly the blocks in each chunk. This technique proved very expensive because of the high cost of sampling from a Binomial distribution, i.e., it takes about 3s in Java and 9s in Mathematica to generate 1,000 binomials.

As future work, we plan to investigate more efficient techniques for generating a fixed set of random numbers from a large interval incrementally.

6.3 Performance

We have implemented our new variant with incremental encoding of files in Java 1.6. The Java Virtual Machine has 1GB of memory available for processing. We report our performance numbers from an Intel Core 2 processor running at 2.16 GHz. Files were stored on a Hitachi 100 GB Parallel-ATA drive with a buffer of 8MB and rotational speed of 7200 RPMs. The average latency time for the hard drive is 4.2ms and the average seek time is 10ms. We use the BSAFE library in Java for implementing the cryptographic operations.

We show in Figure 5 the total encoding time for files of several sizes, divided into several components: Read (time to read the file from disk), PRP (time to compute the two PRPs used in SA-ECC), ECC encode (time to compute error-correcting information for the outer-code), MAC (time to compute a MAC over the file), Challenges (time to compute $q = 1000$ challenges with the inner code), and Encrypt-Write (time to encrypt the error-correcting information and write the parity blocks and the challenges to disk). The results are reported for the $(4096, 1024, 1024)$ inner code and the $(241, 223, 18)$ outer code. We show averages over 10 runs.

The encoding algorithm achieves a throughput of around 3MB/s, and we observe that encoding time grows linearly with file size. The outer error-correcting layer is responsible for most of the encoding overhead (i.e., between 61% and 67% in our tests). The outer code encoding time can be reduced by reducing the outer code distance, at the expense of checking more challenges in Phase I of our framework. We expect that an optimized C implementation of Reed-Solomon encod-

ing would reduce this overhead several times. Among the other components, noticeable overheads are seen in the time to compute a file MAC (around 4.2%), the time to access files from disk (around 11-12%), and the time to encrypt and write to disk the parity blocks (around 15%). Encryption of party blocks is slow because our decoding algorithm demands that each parity block is encrypted separately (if CBC encryption is used, for instance, then a corruption in one parity block will propagate to multiple blocks).

7. CONCLUSIONS

We have proposed in this paper a new framework for theoretical design of POR protocols that incorporates existing POR constructions, and enables design of new protocols with a wide range of parameter tradeoffs. We showed how the protocols of Juels-Kaliski and Shacham-Waters can be simplified and improved using the new framework. We designed a new variant of the Juels-Kaliski scheme that achieves lower storage overhead, tolerates higher error rates, and can be proven secure in a stronger adversarial setting. Finally, we provided a Java implementation of the encoding algorithm of the new variant, in which files are processed and encoded incrementally, i.e., as they are read into main memory.

As future work, we think it is worth exploring further optimizations in our implementation to enhance the encoding throughput. An interesting practical problem is to design different encoding techniques with a minimal number of disk accesses for very large files, i.e., those for which the parity blocks do not fit into main memory. On the theoretical side, we leave open the problems of designing efficient POR protocols that support file updates, as well as publicly verifiable PORs.

8. 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] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession, 2008. IACR ePrint manuscript 2008/114.

[3] M. Bellare and A. Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Proc. CRYPTO '04*, pages 273–289. Springer, 2004. LNCS vol. 3152.

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

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

[6] K. Bowers, A. Juels, and A Oprea. Proofs of retrievability: Theory and implementation, 2008. Available from ePrint, report 2008/175.

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

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

[9] D.L.G. Filho and P.S.L.M. Barreto. Demonstrating data possession and uncheatable data transfer, 2006. IACR eArchive 2006/150. Referenced 2008 at http://eprint.iacr.org/2006/150.pdf.

[10] O. Goldreich. *Foundations of cryptography, Volume I: Basic tools*. Cambridge University Press, 2001. First Edition.

[11] O. Goldreich. *Foundations of cryptography, Volume II: Basic applications*. Cambridge University Press, 2004. First Edition.

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

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

[14] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proc. USENIX Annual Technical Conference, General Track 2003*, pages 29—41, 2003.

[15] M. Luby. LT codes. In *Proc. Symposium on Foundations of Computer Science (FOCS)*, pages 271–Ů282. IEEE, 2002.

[16] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proc. Symposium on Theory of Computation (STOC)*, page 150Ů159. ACM, 1997.

[17] P. Maymounkov. On-line codes. Technical Report TR2002-833, Computer Science Department at New York University, November 2002.

[18] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 573–584, 2005.

[19] J. Patarin. Improved security bounds for pseudorandom permutations. In *Proc. ACM CCS*, pages 142–150, 1997.

[20] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, 1972. Second Edition.

[21] H. Shacham and B. Waters. Compact proofs of retrievability. Proc. Asiacrypt 2008.

[22] M.A. Shah, M. Baker, J.C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest, 2007. Presented at HotOS XI, May 2007.

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