Proofs of Retrievability: Theory and Implementation

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

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

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

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 costs 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. This framework leads to improvements in 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.

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

Key words: storage systems, storage security, proofs of retrievability, proofs of knowledge

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—or needing to know—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 *archived* 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. For example, Sarbanes-Oxley Rule 2-06(a) requires auditors to retain records for a period of seven years. More generally, backup files, while rarely accessed, are of critical importance to all classes of users.

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, bandwidth-efficient construction in which a client stores (or downloads) a tag for the file, as well as some integrity values, but can initiate an unlimited number of challenges. In their construction, the size of the tag multiplied by the size of the integrity values is linear in the file size.

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 propose 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.

Intuition for our scheme.

The PORs we consider operate in essentially the following way: The client applies an error-correcting code ECC_{out} to the target file F to obtain an encoded (expanded) file $\tilde{F}$, which it stores with the server. The code ECC_{out} has the effect of rendering F recoverable even if up to some ϵ-fraction of $\tilde{F}$ is corrupted, where ϵ is a parameter dependent on the choice of ECC_{out}.

To rule out high file-corruption rates and thus ensure that F is retrievable, the client randomly samples $\tilde{F}$. It does so by challenging the server. The client specifies a subset s of blocks in $\tilde{F}$ plus a nonce u (whose purpose we discuss below). The server applies a function $\mathsf{respond}$ to s and returns the result $\mathsf{respond}(s, u)$. With enough challenge rounds, the client obtains the following two-sided guarantee:

1. If the server corrupts more than an ϵ-fraction of $\tilde{F}$, and F is therefore unretrievable, the client will detect this condition with high probability;
2. If the server corrupts less than (or exactly) an ϵ-fraction of $\tilde{F}$, the client is able to retrieve F in its entirety via decoding under ECC_{out}.

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). 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 server via $\mathsf{respond}$.

In most previously proposed POR constructions, the function $\mathsf{respond}$ returns a single file block or an XOR of file blocks. Our key insight in this paper is that $\mathsf{respond}$ may itself apply an *arbitrary error correcting code* . In particular, we consider schemes in which $\mathsf{respond}$ 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]

[1]An "inner code" and "outer code" are the constituents of a *concatenated* error-correcting code. The composition of ECC_{in} and

The inner code ECC_{in}, being computed on the fly by the server over $\tilde{F}$, 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}.

As we show, with an appropriate choice of ECC_{in}, we are able simultaneously to reduce the file expansion in JK and the number of challenges required to ensure a high probability of successful file recovery, that is, to tolerate a higher adversarial error rate ϵ. We achieve all of this in a stronger adversarial model (eliminating the "block isolation" assumption of JK). Similarly, although they do not view it as such, we show that SW implicitly incorporate an erasure code ECC_{in} in their scheme. We describe how to generalize and simplify the SW construction using our framework.

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 original JK protocol in Section 5. In Section 5 we also describe several directions in which the SW construction can be improved using our framework. Finally, we describe several challenges we encountered in implementing the new variant and present performance evaluation in Section 6, and conclude in section 7. In the paper appendix, we describe and prove results regarding "adversarial error-correcting codes," a technical construction employed in our POR protocols.

2. RELATED WORK

The first proposed POR-like construction of which we are aware is that of Lillibridge et al. [13]. 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.

Extending the memory-checking schemes of Blum et al. [5], Naor and Rothblum [19] 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, $\tilde{F}$ effectively includes message authentication codes (MACs) on file segments: 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 non-trivial challenge-response and extraction 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 [11] 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 value and location of each block s_i during file encoding using a pseudorandom function (PRF). Here s_i is a so-called "sentinel" block whose value is independent of the data blocks in F. The other mechanism 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. JK also consider various practical problems associated with error-coding.

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*.) Additionally, Ateniese et al. do not explore the problems around practical integration of their protocol with error-correcting codes, and consequently construct their protocols with the aim of guaranteeing that a server possesses a large fraction of F, but not necessarily F in its entirety.

Shacham and Waters [23] propose a protocol in which blocks of F have associated integrity values. To challenge the server, the client specifies a subset of file blocks. The server returns a digest of the associated integrity values (computed via an algebraic homomorphism), which the client can verify. Block integrity values serve effectively as message authentication codes. Although SW do not state so explicitly, they employ much the same mechanism as Lillibridge et al.: They encode F using an erasure code, which, thanks to the underlying system of MACs, they are able to turn into an error-correcting code. The SW construction embraces a somewhat different design space than JK, and one that is difficult to compare directly—e.g., SW's client key size is rather larger than JK's for natural parameterizations, but JK supports only a limited number of challenge queries.

In other related work, Golle, Jarecki, and Miranov [9] 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 [6] 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 $\mathsf{respond}$ requires computation of a modular exponentiation with respect to a bit-representation of all of F. Shah et al. [24] 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.

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. 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.

ECC_{out} in our scheme is similar in flavor to a concatenated code or a product code [21], although not precisely identical with either.

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

Another challenge in POR construction is that of file updates. In follow-up work to [1], Ateniese, di Pietro, Mancini, and Tsudik [2] describe tools for file updates in PDPs. The lack of error correction and extraction algorithms in basic PDP constructions permit such updates to be performed efficiently, i.e., with minimal communication overhead. Naor and Rothblum support updates in their theoretical model for PORs. In practice, though, updates in *any* POR scheme entail high communication complexity. This is because any change to the contents of file F, no matter how small, must propagate through error-correcting and challenge-response data encoded in $\tilde{F}$. The specification of good batching schemes or other update techniques for PORs 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 $|F| = m$ 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., $l = 256$. The block size may, but need not, correspond with any parameters of the underlying storage medium.) 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 comprises six functions:

$\mathsf{keygen}[\pi] \rightarrow \kappa$: The function keygen generates a secret key κ. (For a public POR, κ may be a public/private key pair.)

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

$\mathsf{challenge}(\eta;\kappa,\omega)[\pi] \rightarrow c$: The function challenge generates a challenge value c for the file η.

$\mathsf{respond}(c,\eta) \rightarrow r$. The function respond–the only one run by the prover—generates a response r to a challenge c.

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

$\mathsf{extract}(\eta;\kappa,\omega)[\pi] \rightarrow F$: The function extract 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; otherwise, it outputs $\perp$.

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 in a challenge and the nonce domain size, respectively. We let C denote the full challenge space $S \times W$.

We model an adversary $\mathcal{A}$ deterministically as a partition (C^+, C^-) of C. C^+ is the set of all challenge values (s,u) to which the adversary responds correctly, while C^- is the set to which it responds incorrectly. As in previous work, we also assume that the adversary is static or memoryless, i.e., that C^+ does not change over time. The reason for this assumption is twofold. First, if C^+ can change, then no meaningful POR is possible: $\mathcal{A}$ can respond correctly when challenged, but stop responding when the client submits the large number of queries required to extract F. Second, if $\mathcal{A}$ can be rewound, it is effectively memoryless. Given a server image from, e.g., a backup tape, rewinding is often achievable in practice.

We let $\epsilon^{\mathcal{A}}_{s,u} = 1$ if $(s,u) \in C^+$, and $\epsilon^{\mathcal{A}}_{s,u} = 0$ otherwise.[3] We let $\epsilon^{\mathcal{A}}_s = \sum_{u \in W} \epsilon^{\mathcal{A}}_{s,u}/w$ be the probability that the adversary responds correctly to challenges on a subset s of blocks for a random choice of u. We let $\epsilon^{\mathcal{A}} = \frac{\sum_{s \in S, u \in W} \epsilon^{\mathcal{A}}_{s,u}}{|C|}$ be the probability that the adversary responds incorrectly to a challenge selected uniformly at random from C, i.e., $\epsilon^{\mathcal{A}} = |C^-|/|C|$.

An ϵ-adversary is one for which $\epsilon^{\mathcal{A}} \leq \epsilon$.

3.3 Our POR framework: Key ideas

We conceptualize a POR in our framework in two phases:

Phase I: Ensuring an ϵ-adversary.

In the first phase of a POR, 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 $\epsilon^{\mathcal{A}} > \epsilon$. This condition implies that F is irretrievable.

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 ρ times, and rejects if any response is incorrect. Otherwise the client accepts.

Assuming that challenge selects $c \in_U C$, i.e., uniformly at random, the probability that an adversary $\mathcal{A}$ is accepted but is *not* an ϵ-adversary, i.e., $\epsilon^{\mathcal{A}} > \epsilon$, is $< \lambda = (1-\epsilon)^\rho$. The value λ can be made arbitrarily small, with an appropriately large ρ.

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 $\tilde{F}$ 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 $c \in_U C$.

SW omits Phase I, i.e., implicitly assumes an ϵ-adversary.

Remark.

In practice, of course, a server may initially be honest, but turn bad at some point and be supplanted 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 $q = 50$, then, the condition $\epsilon^{\mathcal{A}} > \epsilon$ will be detected with probability at least $1 - \lambda$ within 50 days of the server turning adversarial.

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. 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: It relies on the ϵ-bound on $\mathcal{A}$ for successful decoding.[4]

[3] It is, of course, possible to model the adversary probabilistically by associating a probability $\epsilon'^{\mathcal{A}}_{s,u} \in [0,1]$ with (s,u), as in JK. Such a probabilistic adversary can be easily modeled as a deterministic one by letting $\epsilon_{s,u} = 1$ iff $\epsilon'^{\mathcal{A}}_{s,u} \geq 1/2 + \delta$ for some constant δ, and $\epsilon^{\mathcal{A}}_{s,u} = 0$ otherwise. If $\epsilon'^{\mathcal{A}}_{s,u} > 1/2 + \delta$, then it is possible to extract a correct response from $\mathcal{A}$ with overwhelming probability by performing majority decoding on a number of queries polynomial in $1/\delta$. Thus, we can efficiently convert a probabilistic adversary into a deterministic one with negligible error.

[4] There is an intermediate possibility for extraction. A client can

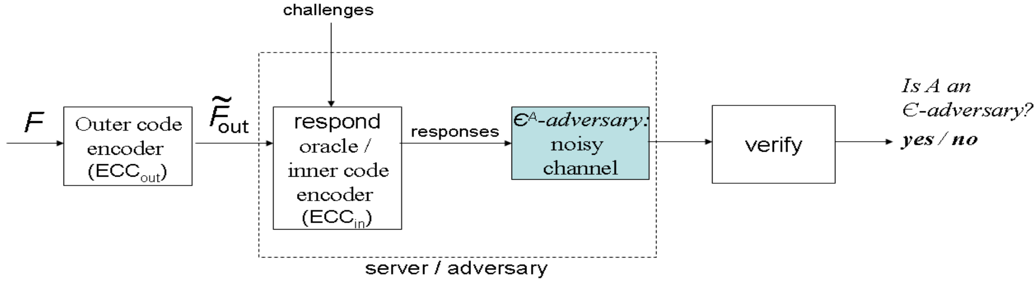

Figure 1: Schematic of Phase I in our POR framework: Testing whether $\epsilon^{\mathcal{A}} \leq \epsilon$, i.e., if $\mathcal{A}$ is an ϵ-adversary.

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. For efficient error-correcting codes and large files, it is usually the case that $k \ll m$. Thus to encode a file of size m we need to resort to a well-known technique called *striping*: The file is partitioned 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 mean implicitly 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. When the client submits a challenge $c = (s, u)$, the respond oracle computes the correct response r. If $\epsilon^{\mathcal{A}}_{s,u} = 1$, then $\mathcal{A}$ corrupts the response in the channel; otherwise, the adversary leaves r unchanged.

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. For the sake of efficiency, the outer code may be an *adversarial error-correcting code* , as defined in Appendix B. Intuitively, an adversarial code uses encryption and permutation to transform a computationally bounded ϵ'-adversary into a random one, i.e., into an adversary that can target its codeword corruptions no better than at random.

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. $\tilde{F}_{out}$ is a component of the full file encoding $\tilde{F}$, which may include supplementary data such as MACs.
2. **Extraction:** If both ordinary downloading of F and error-correction of F_η 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 $\tilde{F}^*_{out}$. W.h.p., this file $\tilde{F}^*_{out}$ is a representation of $\tilde{F}_{out}$ with ϵ'-fraction corruption, where $\epsilon' < \epsilon$. In other words, inner decoding drives down the effective file-corruption rate.
5. **Outer decoding:** The client decodes $\tilde{F}^*_{out}$ under ECC_{out}, obtaining the original file F.

try to download $\tilde{F}_{out}$ and perform error-correction using the outer code only. This approach will work provided that the fraction of corrupted blocks is at most ϵ', as defined below. Thus full extraction is not necessary, for example, when an honest server accidentally corrupts a few blocks.

3.4 Security definition

By viewing a POR as a two-phase process, we are able to offer a security definition that is akin to that of SW and simpler than that of JK. 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,*

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

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. If $d_1 \geq 2\epsilon m$, i.e., the code ECC_{in} is resilient to an ϵ-fraction of file corruption, we could then dispense with the outer code. In practice, constructing an error-correcting code with such a large message size and large distance would be impractical. In constructing a POR, we seek a balance between the resource requirements of the inner code (computation and file-block retrieval) and those of the outer code (file expansion).

4. NEW VARIANT POR PROTOCOL

We describe in this section a new variant of the Juels-Kaliski POR protocol that follows the Phase I / Phase II framework given in the previous section. Our main goals in designing the new variant 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

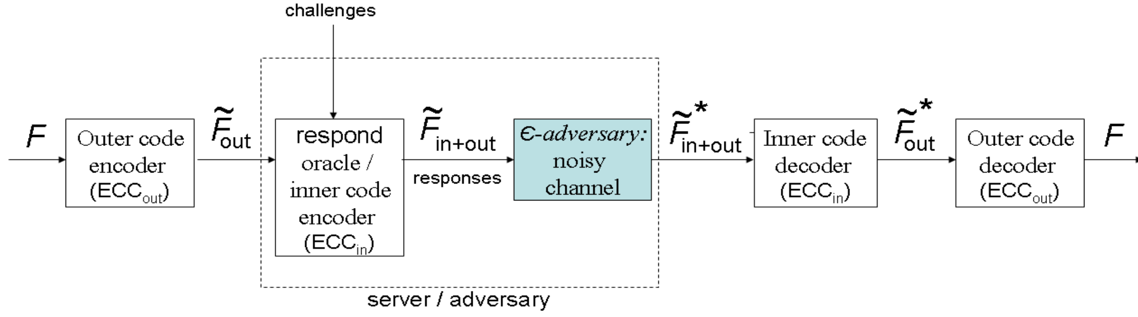

Figure 2: Schematic of Phase II in our POR framework: Extracting F from an ϵ-adversary.

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.

4.1 Details of the Protocol

Building blocks.

To specify the algorithms in our protocol, we need several cryptographic primitives, in particular a symmetric-key encryption scheme ($\mathsf{KGenEnc}, \mathsf{Enc}, \mathsf{Dec}$), a family of pseudorandom permutations $\mathsf{PRP}[n] : \mathcal{K}_{\mathsf{PRP}} \times \{0,1\}^n \rightarrow \{0,1\}^n$ 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 [7, 8].

Preparing challenges.

As described in Phase II of our framework, a challenge is a pair (s, u), such that the response is the u-th symbol in a codeword over an inner coding instance consisting of file blocks from set s. The client stores ciphertexts of some pseudorandomly chosen sets of these challenges and corresponding responses.

Outer layer of error correction.

JK encode algorithm, which has only an outer code, uses an adversarial code. JK use a striped "scrambled" code in which the file is divided into stripes, each stripe is encoded with a standard (n, k, d) Reed-Solomon code and a pseudorandom permutation is applied to the resulting symbols, followed by encryption of permuted file blocks. The permutation and encryption secret keys are known only to the client.

In this paper, we propose and employ a *systematic* adversarial error-correcting code, i.e., one in which the message blocks of F remain unchanged by error-correcting.[5] 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*. Scrambling alone does not ensure a random adversarial channel, as a plaintext file reveals stripe boundary information to an adversary. To hide stripe boundaries from an adversary, i.e., to scramble F, we partition the file into stripes by implicit application of a pseudorandom permutation. (We do not explicitly permute the file.) Our outer code outputs the file F untouched, followed by the "scrambled" parity blocks. To hide stripe boundaries, the parity blocks are then encrypted.

More formally, our adversarial error-correcting code $\mathsf{SA\text{-}ECC}$ takes as input secret keys k_1, k_2 and k_3, and a message M of size m blocks. It encodes via the following operations:

1. Permute M using $\mathsf{PRP}[m]$ with key k_1, divide the permuted 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}.
2. The output codeword is M followed by permuted and encrypted error-correcting information. (The permutation of parity blocks is accomplished by $\mathsf{PRP}[\frac{m}{k}(n-k)]$ with secret key k_2; encryption takes place under key k_3.)

To decode, $\mathsf{SA\text{-}ECC}$ reverses the order of the above operations.

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 $\mathsf{k_{chal}}$, a key $\mathsf{k_{ind}}$ used to sample codeword indices of ECC_{in}, a file MAC key $\mathsf{k^{file}_{MAC}}$, a master encryption key $\mathsf{k_{enc}}$, a file permutation key $\mathsf{k^{file}_{perm}}$, an ECC permutation key $\mathsf{k^{ECC}_{perm}}$ and an ECC encryption key $\mathsf{k^{ECC}_{enc}}$. 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 as follows:

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

In the $\mathsf{challenge}$ algorithm, the client sends to the server j, k^c_j and the random index u derived from $\mathsf{k_{ind}}$. In the $\mathsf{respond}$ algorithm, the

[5] In [12], such a code is briefly described, but not analyzed.

server derives $i_1, \ldots, i_v$ 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 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 exists, 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. Formally, the extract algorithm is as follows:

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 $(\mathsf{k}_{\mathsf{perm}}^{\mathsf{file}}, \mathsf{k}_{\mathsf{perm}}^{\mathsf{ECC}}, \mathsf{k}_{\mathsf{enc}}^{\mathsf{ECC}})$ and obtain F.
3. Compute the MAC over the whole file and check it against the MAC stored with the file. If the MAC verifies, output the file, and otherwise output $\perp$.

4.2 Security analysis

Theorem 1 characterizes the security properties of the new protocol. We give here a brief overview of the security analysis, and defer the full proof to Appendix A.

We assume that our adversary is an ϵ-adversary, i.e., it corrupts a fraction ϵ of the challenge space, and ϵ is smaller than the error correction capability of the inner code. By our majority decoding procedure, a block is not correctly recovered when a large majority of the challenges in $\mathcal{C}$ containing that block is corrupted. We can thus build a mapping between corrupted challenges and corrupted blocks and we can use a simple counting argument to upper bound the fraction of corrupted blocks as a function of ϵ. We also need to ensure that the maximum fraction of block corruptions is within the error correction capability of the outer code.

Once we set the maximum block error rate ϵ', we bound the probability that the file can not be correctly extracted (i.e., $1 - \gamma$) as the probability that at least one of the $\lceil \frac{t}{n} \rceil$ stripes of F has more than $\frac{d_2}{2}$ corruptions (and, thus, exceeds the error correction capability of the outer code). In our decoding algorithm, we pick a set of challenges $\mathcal{C}$ of size $\alpha \frac{t}{v}$. Thus, we have to account in our bound on the (small) probability depending on α that some blocks are not covered in $\mathcal{C}$ (they are treated as erasures in the outer code decoding procedure).

THEOREM 1. *Assume that the encryption scheme is IND-CPA secure, the family* PRP *is pseudorandom and the MAC algorithm is unforgeable. For any $0 < \epsilon \leq \frac{d_1}{2w}$, if there exists an ϵ' for which $0 < \epsilon' < \frac{d_2}{4n} - \frac{e^{-\alpha}}{4}$ and $[1 - (1 - \epsilon')^v](\frac{1}{2} + \delta) = \frac{2\epsilon w}{d_1}$, our scheme is a (ϵ, γ)-secure POR scheme for $\gamma = 1 - \lceil \frac{t}{n} \rceil e^{\frac{d_2}{2} - 2n\epsilon' - \frac{n}{2}e^{-\alpha}} \left(\frac{d_2}{4n\epsilon' + ne^{-\alpha}} \right)^{-\frac{d_2}{2}}$.*

Remark.

For a given fraction ϵ of corrupted challenge, the maximum block error rate ϵ' is determined by the parameters of the inner code, in particular the ratio $\frac{d_1}{w}$. The smaller ϵ', the better our security bound (i.e., γ is closer to 1). For a fixed ϵ', and assuming a fixed codeword size n for the outer code, there are still two degrees of freedom for our security bound. First, the bound gets better when the distance d_2 of the outer code increases, at the expense of an increased storage overhead for the encoded file. Second, we obtain a better bound by increasing α, the average number of decodings for each file block, at the expense of a more expensive decoding algorithm. The reason is that, as α increases, the probability that a certain block is not selected in challenge set $\mathcal{C}$ is reduced, and, thus, the outer code needs to recover from fewer erasures.

4.3 Parameterization

We now explore design tradeoffs achieved by our construction for some example parameter choices. We consider two different classes of inner codes. The first is a class of "theoretical" codes whose existence is guaranteed by the Varsharmov-Gilbert lower bound; these may not be realizable in a computationally efficient sense, but provide a bound on the ideal coding properties of our construction. The second is a class of practical codes easily constructed by concatenation of standard Reed-Solomon codes.

Theoretical codes.

The Varsharmov-Gilbert lower bound on the realizable minimum distance of a code (n, k) is given by the following theorem.

THEOREM 2. *(Varsharmov-Gilbert [21]) It is possible to construct an (n, k) code over an alphabet Σ of size σ with minimum distance at least d, provided that: $\sum_{i=0}^{d-2} \binom{n}{i} (\sigma - i)^i \geq \sigma^{n-k}$.*

We consider several codes over a byte alphabet that follow this lower bound. These codes emerge from 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 a k-symbol message 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. By this procedure, we can obtain 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 [21]. Several codes obtained through concatenation of RSS codes 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'. Under the practical parameterizations we propose for our new variant POR, ϵ' is reduced by at least an order of magnitude compared to ϵ. Thus we can appeal to a more lightweight outer code than the JK scheme, resulting in lower outer-code storage overhead.

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).[6] The graphs in Figure 3 plot 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.

[6]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.

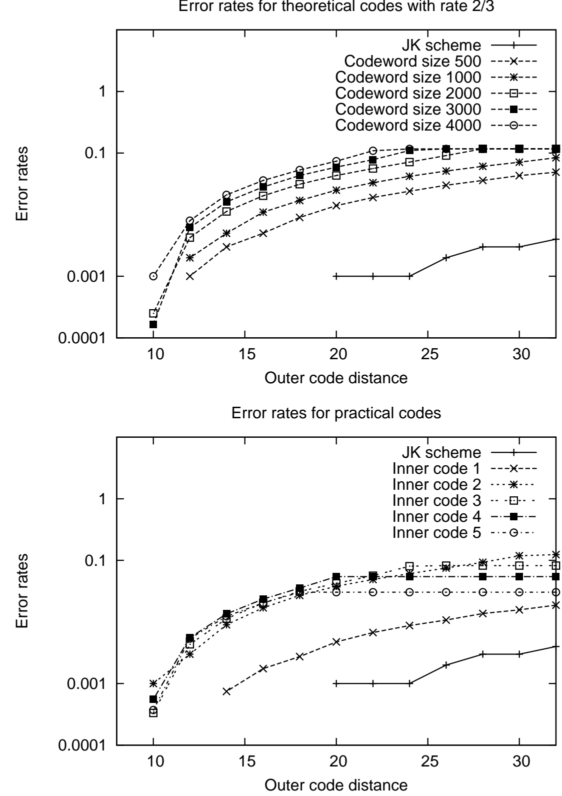

Figure 3: Maximum error rate ϵ tolerated as a function of outer code distances for both theoretical inner codes with rate 2/3 and for different practical codes.

Number of challenges required in Phase I.

Intuitively, as the tolerable adversarial error rate ϵ guaranteed by our error-correcting codes rises, the number of challenges needed in Phase I of our framework decreases. The looser the bound ϵ required on the adversary, the fewer the challenge-response rounds needed to ensure the desired bound.

Figure 4 shows the number of challenges imposed on Phase I for both theoretical codes with rate 2/3 and for 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 quickly becomes prohibitive 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 relative to JK, while checking fewer than 100 challenges in Phase I.

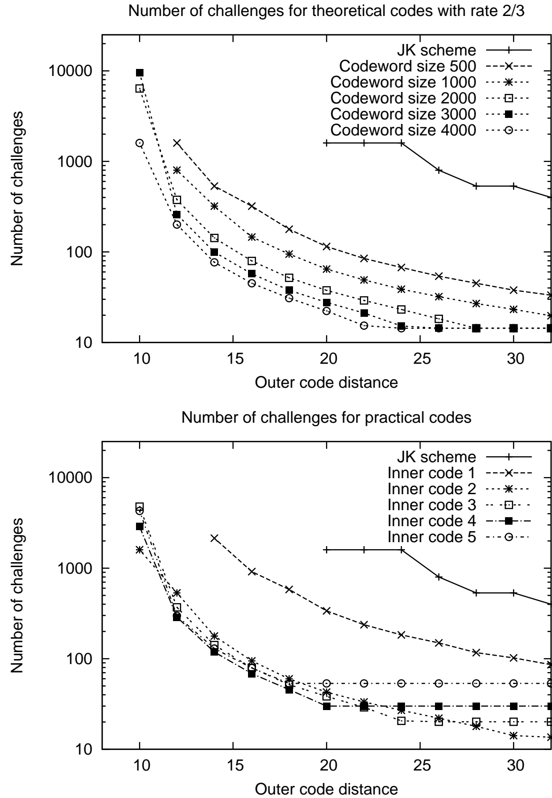

Figure 4: Number of challenges required in Phase I as a function of outer code distance, for both theoretical inner codes with rate 2/3 and for different practical codes.

5. DISCUSSION ON PREVIOUS SCHEMES

5.1 Comparison of New Variant Protocol to Original Juels-Kaliski Scheme

The original JK scheme employs only one outer layer of error-correction. In the challenge phase, the client downloads a number of sentinel values from the server and verifies their correctness. The sentinel values are derived from a pseudorandom seed, so that the client can easily compute the correct values by storing only a small amount of information. Therefore, the challenge phase in the JK scheme 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. Our hope, however, 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 than JK*, since we do not make simplifying assumptions about the adversary's behavior, except for the fact that it is memoryless. JK makes in addition the strong *block isolation assumption*, which stipulates that the probabilities file blocks are returned correctly in a challenge are independent of one another. Our adversary, in contrast, may corrupt each challenge individually, a model supporting inherent correlations among the corruption of file blocks.

5.2 Improvements to Shacham-Waters Scheme

In the SW scheme, the file is divided into n blocks, each of size s sectors. At encoding time, an integrity value σ_i is stored for each block $1 \leq i \leq n$. A file tag of size s is also constructed and needs to be either stored locally by the client, or downloaded from the server at each challenge. In a challenge, the client sends indices for a subset of blocks and receives a digest of the block values. Due to a homomorphic MAC construction, the block integrity values can be combined into a single digest value, which effectively reduces the challenge bandwidth in the SW protocol.

The construction implicitly assumes an ϵ-adversary (in which case the client could perform an unlimited number of challenges), and, thus, omits phase one in our framework. However, the scheme setup allows us easily to extend it to include the first phase. Similarly to our scheme, we could append encrypted integrity values, each computed over a subset of file blocks. To ensure an ϵ-adversary, the client could then verify several of the appended values.

Since the scheme essentially employs a MAC for each block, the blocks whose MAC does not verify can be treated as erasures, and, thus, the scheme could use an erasure code in the outer layer instead of a more expensive error-correcting code.

Using our conceptual framework, we could generalize and simplify the SW scheme in several respects.

1. To extract the file, the construction implicitly uses an inner erasure code manually crafted by SW. The decoding cost of this erasure code is $O(n^2 s + n^3)$, with $ns = |F|$. SW allows different parameterizations to tradeoff the tag size versus the size of integrity values. In order to obtain a linear decoding cost, we need to set $n = 1$ and $s = |F|$, meaning that the whole file is processed as a single block and the tag size becomes linear in $|F|$. This parameterization, though, is not acceptable in practice, as it defeats the purpose of a POR protocol. For a more natural choice of $n = s = \sqrt{|F|}$, the decoding cost of SW becomes $O(|F|\sqrt{|F|})$.
We could generalize the SW construction by replacing this ad-hoc erasure code with an erasure code with linear encoding and decoding time, e.g., Tornado codes [16], on-line codes [17], LT codes [15] and Raptor codes [25]. 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 would benefit in simplifying the security proofs of SW. For example, the proofs from Section 4.2 in [23] could be easily inferred from the decoding properties of the above mentioned erasure codes.
2. The SW construction in the public-key setting is computationally expensive, as the number of exponentiations (performed in a large group) to check a challenge over a subset of blocks is proportional to the sector size s. One solution to reduce the computational overhead for verifying challenges

is to replace their extraction based on an inner erasure code and MAC verification with an error-correcting code, similarly to our construction. Their homomorphic MACs could still be used in the outer layer with an outer erasure code. This idea would make the public-key construction feasible in a practical implementation.

Compared with our new variant, in the SW scheme there are different tradeoffs between parameters. Since the SW paper does not contain a detailed discussion about parameter choices, it is quite difficult to exactly compare the tradeoffs in the two schemes.

6. IMPLEMENTATION

In this section, we describe several challenges we encountered in implementing our proposed POR.

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" pseudrandom 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}, the equivalent of a cipher with a 27-bit block size.

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.

Patarin [20] offers a construction with a stronger bound. He 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 order of magnitude slower than having the tables stored in main memory.

Incremental encoding.

Our implementation supports encoding of files whose size may considerably exceed that of client main memory. (This is not uncommon for large archival files.) In such cases, random accesses to file blocks as prescribed in the basic JK algorithm are impractical. Most of F will reside on disk, and random accesses to disk blocks are slow. For example, a Hitachi 100 GB Parallel-ATA drive used in our experiments has an average seek time of 10ms. Random access to every block of a 1GB file with 32-byte blocks on this hard drive would require some 66,000s, over eighteen hours.

For this reason, our implementation involves new techniques for POR encoding that avoid the need for random accesses and instead encode the F *in one pass*. One-pass processing poses a particular challenge in computation of the outer error-correcting code. The adversarial code SA-ECC constructs stripes as random blocks drawn from all of F. Naïve, sequential error encoding of stripes would therefore require random accesses across F.

The approach we have adopted is *incremental* computation of codewords over the component stripes of F. We load F into main memory in a sequence of consecutive segments $F^{(1)}, F^{(2)}, \ldots, F^{(z)}$ and process each segment individually in turn. In doing so, we accumulate partial computations of parity blocks for all stripes in F. For example, suppose that the message $M =< M_1, M_2, \ldots, M_k >$ representing a given stripe across F is to be mapped into n-symbol parity-block vector PB using linear error-correcting code ECC. A given segment $F^{(j)}$ of F, when loaded into main memory, will generally include only a subset of symbols in M. We may express this subset as a k-symbol vector $M^{(j)}$ in which each block M_i absent from $F^{(j)}$ is represented by a 0. Observe that $\mathsf{ECC}(M) = \sum_j \mathsf{ECC}(M^{(j)})$ (for vector addition over the underlying field). Thus we can accumulate partial parity-block computations over file segments. That is, we initialize $PB = 0$. On loading file segment $F^{(j)}$, we perform the update $PB += \mathsf{ECC}(M^{(j)})$. After the last segment $F^{(z)}$ is processed, the vector PB will contain the correct parity blocks for M.

A similar challenge arises in the precomputation of our q challenge-response pairs. The set of blocks s in a challenge $c = (s, u)$ is drawn uniformly (pseudorandomly) from the blocks of F. To avoid random accesses to file blocks, we have implemented the brute-force approach of precomputing and storing in main memory all v indices of the blocks in s. We then compute the corresponding response r incrementally across file segments.[7]

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 a Dell Latitude D620 running a 2.16 GHz Intel Core 2 processor. 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 RSA BSAFE Java library for the cryptographic operations.

We show in Figure 5 the total encoding time for files of several sizes, divided into several components: Read (the time to read the file from disk), PRP (the time to compute the two PRPs used in SA-ECC), ECC encode (the time to compute error-correcting information for the outer-code), MAC (the time to compute a MAC over the whole file), Challenges (the time to compute $q = 1000$ challenges with the inner code), and Encrypt-Write (the time to encrypt the error-correcting information and write the parity blocks and challenges to disk). Results are reported for a $(4096, 1024, 1024)$ inner code and $(241, 223, 18)$ outer code. For these parameters, the largest file size that can be encoded incrementally is about 12GB. (This bound is dictated by main-memory storage requirements for the incremental computations.) We show averages over 10 runs.

The encoding algorithm achieves a throughput of around 3MB/s, and the encoding time grows roughly 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 dis-

[7] We have also investigated an incremental technique for sampling v challenge blocks uniformly across F. Let v be the number of blocks not yet sampled upon loading of the file segment $F^{(j)}$. We determine the number of challenge blocks in a given file segment $F^{(j)}$ via pseudorandom sampling from the binomial distribution $B(v, p)$, where $p = 1/(zj + 1)$, and then select these blocks uniformly (pseudorandomly) from $F^{(j)}$. Sampling from a binomial distribution is itself computationally intensive, however, requiring about 3ms in Java and 9ms in Mathematica in our experiments.

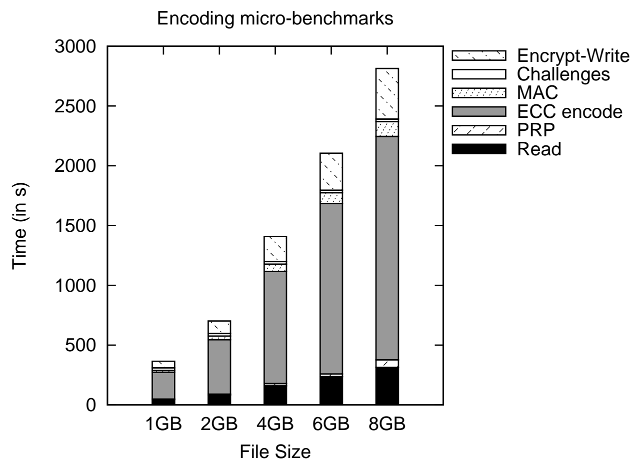

Figure 5: Encoding micro-benchmarks for $(4096, 1024, 1024)$ **inner code and** $(241, 223, 18)$ **outer code.**

tance, at the expense of checking more challenges in Phase I of our framework. We expect that an optimized C implementation of Reed-Solomon encoding, rather than Java, would reduce this overhead by about a factor of three. Parallelization across processors would lead to further improvement—roughly a doubling of performance on the Intel Core 2. Other 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 construction requires a blockwise cipher mode. A chained mode like CBC would cause errors to propagate across the file, frustrating the error-correction process. Thus we make use of 256-bit AES encryption in ECB mode, deriving a fresh key for each block via a PRF on a master key XORed with the block index. (In our next version of the prototype, we plan to implement a standard tweakable block cipher such as XEX-AES [22].)

7. CONCLUSION

We have proposed a new architectural framework for POR protocols that encompasses existing protocols, but also extends into an improved design space. 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 exceeding main memory size are loaded from disk, processed, and encoded incrementally.

In future work, we plan to explore further implementation optimizations. We believe that there is ample room to improve the resource-dominant outer coding step. The problem of precomputing challenge/response values efficiently is also an important one, as is that of finding more efficient adversarial codes, perhaps based in previous literature on MACs plus erasure codes. On the theoretical side, there remain the open problems of designing efficient POR protocols with file updates and 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 *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 *CRYPTO '04*, pages 273–289. Springer, 2004. LNCS vol. 3152.

[4] J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *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] 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.

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

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

[9] 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.

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

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

[12] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files, 2008. ACM CCS slide presentation. Referenced 2008 at http://www.rsa.com/rsalabs/node.asp?id=3357.

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

[14] R. J. Lipton. A new approach to information theory. In *11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 699–708, 2004.

[15] M. Luby. LT codes. In *FOCS*, pages 271–282. IEEE, 2002.

[16] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *STOC*, pages 150–159. ACM, 1997.

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

[18] S. Micali, C. Peikert, M. Sudan, and D. Wilson. Optimal error correction against computationally bounded noise. In *TCC*, pages 1–16. Springer, 2005. LNCS vol. 3378.

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

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

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

[22] 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.

[23] H. Shacham and B. Waters. Compact proofs of retrievability, 2008. IACR ePrint manuscript 2008/073.

[24] 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.

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

APPENDIX

A. SECURITY PROOF

In this appendix, we present the proof of Theorem 1. For simplification, we assume that the cryptographic primitives are ideal, i.e., ciphertexts reveal no information about plaintexts, the output of a pseudorandom permutation from family PRP is random, and an adversary is unable to forge a MAC for a new message. Let $0 < \epsilon \leq \frac{d_1}{2w}$ and assume that the client is dealing with an ϵ-adversary. We bound the probability that the file is not correctly extracted, after the client performs the extract algorithm using a set of challenges $\mathcal{C}$ of size $N_{\mathcal{C}}$. We proceed in two steps, following the two phases of the extract algorithm.

Decoding using the inner error-correcting code.

After the first phase of extract the probability that $\mathcal{D}_i = \Phi$ for a given block i is bounded by $\left(1-\frac{v}{t}\right)^{N_{\mathcal{C}}} = \left(1-\frac{v}{t}\right)^{\alpha\frac{t}{v}} \approx e^{-\alpha}$. We call such a block an *erasure*. We bound below the probability that the client outputs an incorrect decoding for a given block, assuming that the decoding set for that block is not empty.

First, we define a *corrupted challenge* $(i_1,\ldots,i_r)$ in space $\mathcal{C}$ to be such that at least $\frac{d_1}{2}$ out of the w challenges $(i_1,\ldots,i_r,u)$, $u \in [1,w]$ are corrupted. A corrupted challenge adds an incorrect decoding to each set $\mathcal{D}_{i_s}$, for $s \in [1,r]$ in the extract algorithm. Let us bound the number of such corrupted challenges, denoted $\mathsf{T}_{\mathsf{corr}}$.

If we denote by $\epsilon_c = \sum_{u=1}^{w} \epsilon_{cu}$ the number of corrupted challenges in space $\mathcal{C} \times [1,w]$ containing c, then a challenge c is corrupted in space $\mathcal{C}$ if $\epsilon_c > \frac{d_1}{2}$. If $\mathsf{T}_{\mathsf{corr}}$ challenges are corrupted in space $\mathcal{C}$, it follows that at least $\frac{d_1}{2}$ challenges are corrupted in space $\mathcal{C} \times [1,w]$. Then

$$\frac{d_1}{2}\mathsf{T}_{\mathsf{corr}} \leq \sum_{c\in\mathcal{C}} \epsilon_c = \epsilon w N_{\mathcal{C}},$$

and $\mathsf{T}_{\mathsf{corr}} \leq \min(N_{\mathcal{C}}, \frac{2\epsilon w N_{\mathcal{C}}}{d_1}) = \frac{2\epsilon w N_{\mathcal{C}}}{d_1}$, since $\epsilon \leq \frac{d_1}{2w}$.

Since we use majority decoding for extraction, to corrupt a block in the first step of extract, the adversary must corrupt at least a fraction $\frac{1}{2}+\delta$ of all challenges in space $\mathcal{C}$ that contain that block. The adversary's goal is to corrupt as many blocks as she can while corrupting few enough challenges to go undetected by the client. However, since we force the adversary to reply incorrectly to at most a fraction ϵ of all challenges, we can give an upper bound on the fraction ϵ' of all file blocks that the adversary can corrupt.

Let ϵ' be the fraction of blocks corrupted by the adversary. Thus, out of the t file blocks, $z=\epsilon' t$ are corrupted (or black) and $t-z$ are correct (or white). The adversary's strategy is to answer incorrectly a sufficient fraction (i.e., $\frac{1}{2}+\delta$) of the challenges that contain at least one black block. Given that in the extraction algorithm we choose $N_{\mathcal{C}}$ challenges uniformly at random from the challenge space, we can compute the expected number of challenges that need to be corrupted in the extraction algorithm.

Let X_i be an indicator random variable taking value 1 if challenge i has at least one black block, and value 0, otherwise. Then $\Pr[X_i = 1] = 1-\left(1-\frac{z}{t}\right)^r$. If $X = (\frac{1}{2}+\delta)\sum_{i=1}^{N_{\mathcal{C}}} X_i$, then X is an upper bound on the number of corrupted challenges in the extraction algorithm's first phase. We can compute $E(X) = N_{\mathcal{C}}[1-(1-\epsilon')^v](\frac{1}{2}+\delta)$.

But the expected number of challenges answered incorrectly by the adversary in the extraction algorithm needs to be smaller than the number of challenges that are corrupted in space $\mathcal{C}$:

$$N_{\mathcal{C}}[1-(1-\epsilon')^v](\frac{1}{2}+\delta) = E(X) \leq \mathsf{T}_{\mathsf{corr}} \leq \frac{2\epsilon w N_{\mathcal{C}}}{d_1}.$$

It follows that ϵ' satisfies $[1-(1-\epsilon')^v](\frac{1}{2}+\delta) \leq \frac{2\epsilon w}{d_1}$. In the worst-case, the adversary corrupts the maximum number of file blocks, thus we consider the maximum ϵ' that satisfies the previous inequality, i.e., the one for which $[1-(1-\epsilon')^v](\frac{1}{2}+\delta) = \frac{2\epsilon w}{d_1}$. Following the theorem conditions, we assume that there exists such an ϵ' for which, in addition, $0 < \epsilon' < \frac{d_2}{4n} - \frac{e^{-\alpha}}{4}$.

Decoding using the outer layer code.

In the second phase, the client confronts an adversary that corrupts a fraction ϵ' of the file blocks and the "scrambled" parity blocks of the inner error-correcting code. In addition, a (tiny) fraction of blocks are not covered in the challenge set constructed in the first phase. We treat these blocks as erasures. Our goal now is to bound the probability that the file cannot be correctly extracted in the second phase.

F' (a codeword in the outer code) consists of the original file F of size m, and the permuted and encrypted error-correcting information, of size $t-m$. Assuming a length-preserving encryption scheme is used for encrypting the error-correcting information, we know that the number of stripes is equal to $\lceil \frac{t}{n} \rceil = \lceil \frac{m}{k} \rceil$. Let R be a stripe of n blocks in file F'. Stripe R consists of k blocks in the original file (the first m blocks of F') and $n-k$ blocks of error-correcting information (the last $t-m$ blocks of F').

We bound the probability that stripe R can not be fully extracted. Each block in the stripe is either an erasure or has been reconstructed in phase one. There is a small probability, bounded by $e^{-\alpha}$, that a given block is not included in the challenge set $\mathcal{C}$, and, thus, is an erasure. Let $Y_1,\ldots,Y_n$ be indicator random variables for block erasures in stripe R. We define $Y_i=1$ if block i from stripe R is not included in the challenge set in phase one, and $Y_i = 0$, otherwise, for $i \in [1,n]$. Then, $\Pr[Y_i=1] \leq e^{-\alpha}$ and, if $Y = \sum_{i=1}^n Y_i$, $E(Y) \leq ne^{-\alpha}$.

We know that the adversary corrupts at most $z = \epsilon' t$ of the t blocks in file F', but we do not know how the adversary splits the corrupted blocks between the original file (the *left* side) and the error-correcting information (the *right* side). We can, though, upper bound the number of corrupted blocks on each side with z.

Let $Z_1,\ldots,Z_k$ be indicator random variables for corruptions in stripe R on the left side, and $Z_{k+1},\ldots,Z_n$ indicator random variables for corruptions in stripe R on the right side. We define $Z_i = 1$ if block i from stripe R is corrupted in phase one, and $Z_i = 0$ otherwise, for $i \in [1,n]$. Since the left and right sides of the file are permuted with a secret key, we know that the adversary has no better chance than spreading the corruptions in each side independently. Thus, $\Pr[Z_i=1] \leq \frac{z}{m}$, for $i \in [1,k]$ and $\Pr[Z_i = 1] \leq \frac{z}{t-m}$, for $i \in [k+1,n]$. Moreover, we can upper bound Z_i with independent Bernoulli random variables, whose sum has mean $k\frac{z}{m} + (n-k)\frac{z}{t-m} = \frac{2z}{\lceil \frac{t}{n} \rceil} = 2n\epsilon'$. Denote $Z = \sum_{i=1}^n Z_i$. Then $E(Z) \leq 2n\epsilon'$.

Stripe R is corrupted if $Y + 2Z$ exceeds the distance d_2 of the outer layer error-correcting code. We apply Chernoff bounds to bound the probability that stripe R can not be correctly extracted:

$$\Pr\Big[\frac{Y}{2} + Z > \frac{d_2}{2}\Big] \quad \leq \quad \Pr\Big[\frac{Y}{2} + Z > (1+\delta_2)(2n\epsilon' + \frac{n}{2}e^{-\alpha})\Big]$$

$$
\begin{aligned}
&< e^{\frac{d_2}{2}-2n\epsilon'-\frac{n}{2}e^{-\alpha}}\left(\frac{d_2}{4n\epsilon'+ne^{-\alpha}}\right)^{-\frac{d_2}{2}} \\
&= B,
\end{aligned}
$$

for $\delta_2 = \frac{d_2}{4n\epsilon'+ne^{-\alpha}} - 1 > 0$.

Now, the probability that extract is not successful is bounded by the number of stripes in F' times the probability that a given stripe is not extracted successfully, which is bounded by $\lceil\frac{t}{n}\rceil B$. This effectively proves that our system is an (ϵ, γ) POR proof for $\gamma = 1 - \lceil\frac{t}{n}\rceil B$. ∎

B. ADVERSARIAL ERROR-CORRECTING CODES

The use of cryptography to improve resilience in error-correcting codes has seen limited attention in the literature. Lipton [14] first proposed a model of computationally bounded channels for error-correcting codes. In an unpublished manuscript, Gopalan, Lipton, and Ding [10] proposed the technique of "code scrambling," whereby a codeword is permuted and encrypted under a secret key to reduce a computationally-bounded adversarial noisy channel to a random one. (JK uses essentially this idea, with codeword striping.) Stronger results were obtained by Micali, Peikert, Sudan, and Wilson [18], who proposed a public-key construction in which messages digitally signed; signature verification permits a receiver to pick out correct messages after list decoding. Their construction can correct an error rate of $1/2 - \gamma$ for any constant $\gamma > 0$ over binary channels (double the classical bound).

These previous papers have proposed specific constructions and analyzed their error resilience asymptotically with respect to the code rate. Our goal in this paper is different. We do not look to computational assumptions on the channel to achieve better error-correcting bounds than would otherwise be possible. Rather, we use cryptography to construct more *practical* codes within classical error-correcting bounds. Thus we are interested in bounds concretely derived from code parameters. Also, unlike previous work, we consider error-correction with a non-negligible probability of decoding failure. (We treat probabilities asymptotically in cryptographic key lengths, though.) Thus we require a new definition.

Recall that in our POR construction, we apply an adversarial error-correcting code exclusively in the outer code. The inner code uses an ordinary error-correcting code.

Definition.

We define an (n, k)-*adversarial error-correcting code* AECC as a public key space PK, a private key space SK, an alphabet Σ, and a triple of functions:

- A probabilistic function keygen : $1^l \rightarrow K = (sk, pk) \in SK \times PK$;
- a deterministic function encode : $K \times \Sigma^k \rightarrow \Sigma^n$;
- and a deterministic function decode : $K \times \Sigma^n \rightarrow \Sigma^k$.

A *secret-key* adversarial code is one in which $PK = \phi$. A *public-key* adversarial code is one in which encode gets secret input ϕ.

Letting Γ denote the set of outputs generated by the oracle encode, we define the following experiment:

Experiment $\mathbf{Exp}_{\mathcal{A},\mathsf{AECC}}[l]$
  $K = (sk, pk) \leftarrow \mathsf{keygen}(1^l)$;
  $(c, c') \leftarrow \mathcal{A}^{\mathsf{decode}(K,\cdot),\mathsf{encode}(K,\cdot)}(pk)$;
  if $\mathsf{decode}(K, c) \neq \mathsf{decode}(K, c')$ then output $|c - c'|$
    else output ∞

In this experiment, the adversary may experiment arbitrarily with the oracle decode, but does not have direct access to sk. The output is the Hamming distance between a pair of codewords (c, c') selected by $\mathcal{A}$ that decode to different messages.
The aim of the adversary in our security definition now is to find such a pair of codewords (c, c') with minimized $|c - c'|$.

DEFINITION 2. AECC *is a* (β, δ)-bounded *adversarial error-correcting code if* $\Pr[\mathbf{Exp}_{\mathcal{A},\mathsf{AECC}}[l] \leq \beta n] - \delta$ *is negligible in* l *for any adversary* $\mathcal{A}$ *polynomially bounded in* l.

Remarks.

- Against a computationally unbounded adversary, of course, an adversarial error-correcting code is no stronger than its underlying error-correcting code, i.e., permutation and encryption do not create greater resiliency.
- For our POR construction, this definition is slightly stronger than we require. In a POR, the target codeword c is output by the user / verifier who stores $\tilde{F}$. That is, it must be output by the oracle decode, rather than constructed existentially by $\mathcal{A}$.
- An interesting problem seems to arise when sk is made public, i.e., when there are no secret keys. Is it possible for such an "open" adversarial code to correct an error rate ϵ that exceeds the correctable error rate for an unbounded channel?

Constructions.

We first prove that code SA-ECC described in Section 4.1 is secure according to our definition.

PROPOSITION 1. *With the parameters from Theorem 1, code* SA-ECC *is an* $(2\epsilon' + \frac{e^{-\alpha}}{2}, B)$ *adversarial code.*

Proof: Let $\beta = 2\epsilon' + \frac{e^{-\alpha}}{2}$ and assume that the adversary in experiment $\mathbf{Exp}_{\mathcal{A},\mathsf{AECC}}$ from Definition 2 outputs two codewords c and c' at distance at most βt. It follows that there is at least one stripe for which the corresponding codewords in code ECC_{out} are at distance at most $\frac{\beta t}{\lceil\frac{t}{n}\rceil} = \beta n$. We know that $\beta n < \frac{d_2}{2}$ from the conditions of the theorem, and this implies that the stripe can not be successfully recovered. But we have bounded in Theorem 1 the probability of unsuccessful decoding of a given stripe by B. This proves that the outer layer code is an $(2\epsilon' + \frac{e^{-\alpha}}{2}, B)$ adversarial error-correcting code. ∎

Besides the adversarial code SA-ECC proposed here, there are other adversarial code constructions in the literature. By applying integrity protection to segments of F, it is possible in effect to convert an ordinary erasure code into an adversarial error-correcting code. Lillibridge et al. [13] and Naor and Rothblum [19] effectively propose the application of MACs to file blocks. This approach can yield an adversarial error-correcting code with a reduced rate.

The special innovation underlying the outer code in SW is what might be viewed as a "homomorphic" MAC scheme, one for which the set of MACs over b blocks is $o(b)$ (at the expense of a somewhat larger secret key size). This MAC scheme is coupled with an erasure code to achieve an adversarial error-correcting code. SW also propose a public-key adversarial error-correcting code based on aggregate signatures.